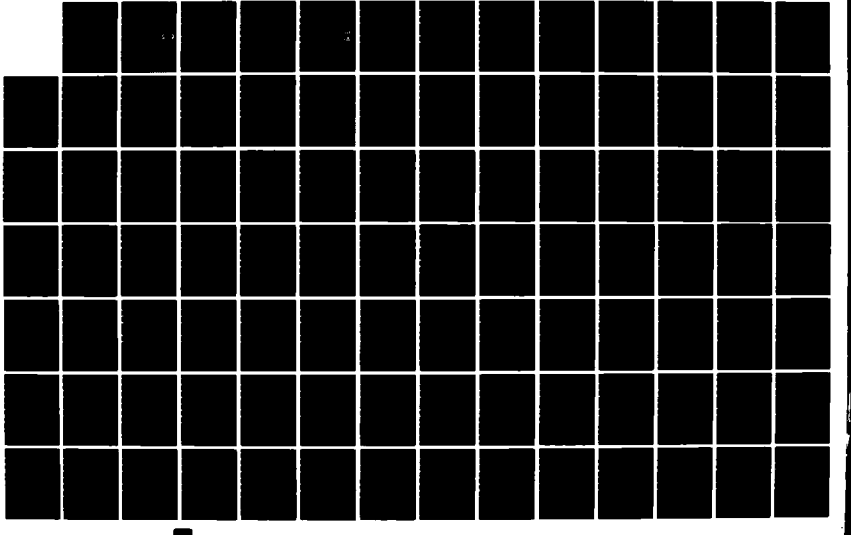
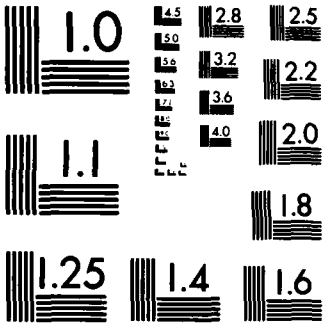
AD-A158 034 USING SOFTWARE TECHNOLOGY TO SPECIFY ABSTRACT INTERFACES IN VLSI DESIGN(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH R R GROSS 1985 1/2

UNCLASSIFIED AFIT/CI/NR-85-92D F/G 9/5 NL





UNCLASS

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(1)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/CI/NR 85-92D	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Using Software Technology to Specify Abstract Interfaces in VLSI Design		5. TYPE OF REPORT & PERIOD COVERED THESIS/DISSERTATION 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Richard Rutter Gross		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS AFIT STUDENT AT: University of North Carolina Chapel Hill		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS AFIT/NR WPAFB OH 45433 - 6583		12. REPORT DATE 1985 13. NUMBER OF PAGES 150
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASS 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

DTIC ELECTE AUG 19 1985 S D B

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

APPROVED FOR PUBLIC RELEASE: IAW AFR 190-1

5 AUG 1985

Lynn Wolaver
LYNN E. WOLAVER
Dean for Research and
Professional Development
AFIT, Wright-Patterson AFB OH

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

ATTACHED

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASS

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AD-A158 034

DTIC FILE COPY

85 8 13 043

RICHARD RUTTER GROSS. Using Software Technology to Specify Abstract Interfaces in VLSI Design. (Under the direction of PETER CALINGAERT.)

ABSTRACT

Good techniques for VLSI design change management do not now exist. A principal reason is the absence of effective methods for the specification of abstract interfaces for VLSI designs. This dissertation presents a new approach to such specification, an approach that extends to the VLSI domain D.L. Parnas's techniques for precise specification of abstract software design interfaces to the VLSI domain. The proposed approach provides important new features, including scalability to VLSI design levels, integration into the design life-cycle, and a uniform treatment of functional, electrical, and geometric design information. A technique is also introduced for attaching a value to the degree of specification adherence of a candidate module.

To illustrate and evaluate the proposed approach, an example specification method using it is described and evaluated experimentally.

AFIT RESEARCH ASSESSMENT

The purpose of this questionnaire is to ascertain the value and/or contribution of research accomplished by students or faculty of the Air Force Institute of Technology (AU). It would be greatly appreciated if you would complete the following questionnaire and return it to:

AFIT/NR
Wright-Patterson AFB OH 45433

RESEARCH TITLE: Using Software Technology to Specify Abstract Interfaces in VLSI Design

AUTHOR: Richard Rutter Gross

RESEARCH ASSESSMENT QUESTIONS:

1. Did this research contribute to a current Air Force project?

( ) a. YES ( ) b. NO

2. Do you believe this research topic is significant enough that it would have been researched (or contracted) by your organization or another agency if AFIT had not?

( ) a. YES ( ) b. NO

3. The benefits of AFIT research can often be expressed by the equivalent value that your agency achieved/received by virtue of AFIT performing the research. Can you estimate what this research would have cost if it had been accomplished under contract or if it had been done in-house in terms of manpower and/or dollars?

( ) a. MAN-YEARS _____ ( ) b. $_____

4. Often it is not possible to attach equivalent dollar values to research, although the results of the research may, in fact, be important. Whether or not you were able to establish an equivalent value for this research (3. above), what is your estimate of its significance?

( ) a. HIGHLY SIGNIFICANT ( ) b. SIGNIFICANT ( ) c. SLIGHTLY SIGNIFICANT ( ) d. OF NO SIGNIFICANCE

5. AFIT welcomes any further comments you may have on the above questions, or any additional details concerning the current application, future potential, or other value of this research. Please use the bottom part of this questionnaire for your statement(s).

NAME GRADE POSITION

ORGANIZATION LOCATION

STATEMENT(s):

FOLD DOWN ON OUTSIDE - SEAL WITH TAPE

AFIT/NR
WRIGHT-PATTERSON AFB OH 45433

OFFICIAL BUSINESS
PENALTY FOR PRIVATE USE. $300

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 73236 WASHINGTON D.C.

POSTAGE WILL BE PAID BY ADDRESSEE

AFIT/ DAA
Wright-Patterson AFB OH 45433

FOLD IN

ACKNOWLEDGMENT

J.S. Bach acknowledged his work with the Latin phrase *Soli Deo Gloria* ("Glory to God alone"). If he, being great, made such an acknowledgment, how can I or anyone else do other?

One especially noteworthy area of God's help to me has been His provision of a supremely patient and encouraging wife and family. Without them, I could certainly never have finished this work.

I am also grateful to have been able to participate, while this dissertation work was being conducted, in a collateral research project funded by Semiconductor Research Corporation Grant No. 41258. This dissertation has greatly benefited from the discussions I had with the other participants in that project.

CONTENTS

USING SOFTWARE TECHNOLOGY TO SPECIFY ABSTRACT INTERFACES IN VLSI DESIGN

by

Richard Rutter Gross

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill

1985

Approved by:

Peter Calingaert
Advisor

Kye S. Hedlund
Reader

Fredrick P. Brooks
Reader

The views expressed in this dissertation are those of the author and do not reflect the official policy or position of the Department of Defense of the U.S. Government.

CHAPTER 1

INTRODUCTION

1.1 Definition of the Problem.

VLSI design is integrated circuit design in which brute force no longer works. The major concern in VLSI design is the management of complexity [Mudg81, Séqu83], not just putting together a system with whatever means are handy.

Traditional techniques for complexity management, such as hierarchy, restriction, and structuring, have been applied in the VLSI context: nevertheless, the design process is still costly, and hundreds of designer-years are being invested in the development of state-of-the-art VLSI circuits [Latt81, Cane83]. The unconstrained nature of the VLSI design medium leads to some of this cost [Séqu83], in that only inefficient algorithms are available to apply to the typically NP-hard problems, such as one-dimensional placement [Valt82] and optimal routing [John82], encountered in design construction and verification. Nevertheless, while such traditional costs of VLSI design are still the subject of much research, another source of cost, only lately recognized, is becoming a major concern among designers. This cost arises from the ripple effect of changes during the design process [Wern83], and its seriousness stems from its particular sensitivity to the increases in complexity that characterize modern VLSI design. Belady and Lehman's work [Bela79], for example, suggests that for software systems "increasing system complexity leads to a regenerative, highly non-linear, increase in the effort and cost of system maintenance and also limits ultimate system growth." While similar studies, to my knowledge, have not yet been directed specifically at VLSI systems evolution, there is good reason to suspect that the effects of progressive changes on VLSI systems are comparable.

Furthermore, this "cost of change" compounds in the following way. Competitive pressures for denser, more capable, and hence more complex circuits beget increased refinement of designs, or increased change. Such increased change, Werner notes, is necessitated by elevated performance standards for modern chips, possibly even requiring retrofitting a design in progress to include new technologies or capabilities. However, the same competitive pressures also demand early production of these more-complex chips, so that larger teams, partitioning the design task, are assembled to meet delivery schedules accelerated by intense competition. Increasing complexity thus has two effects: (1) more change; and (2) larger design teams. As Brooks [Broo75] notes,

either effect alone increases the amount of *communication* required in the design project, and the cost of this communication must be added to the amount of design work to be done. The combination of these effects compounds costs of communication and thus of design, making the cost factors of design in the multi-designer environment significantly different from those that have been traditionally applied.

Consequently, as VLSI circuits grow larger, the cost of change management, especially in the now-typical multi-person design effort, may well become the primary concern in the VLSI design process. Because current VLSI design techniques focus primarily on developing correct initial designs and not yet on the management of design change, the development and study of VLSI design change management techniques are timely and important.

1.2 Potential Contribution from Software Engineering Techniques.

There have been numerous recent suggestions (for example, see [Rade82, Kuni84, Musa85]) that transfer of design technology from software engineering, which has dealt with systems with similar numbers of components for at least a decade, might either improve the theoretical base for VLSI design or at least enhance the effectiveness of emerging VLSI design methodologies. Séquin [Séqu83] supports this suggestion by listing the following software design characteristics that possess counterparts in VLSI design.

a. Appropriate design representations are crucial to design success.
b. Abstraction provides an appropriate vehicle for dealing with complexity. The possibility of exploiting such abstractions, using high-level languages, is promising.
c. At the lowest level, on the other hand, the design may be restricted to a limited set of well-understood constructs.
d. The design task can be partitioned and structured. For example, functional design can be separated from implementation.
e. Testing of the final realization against a well-defined system specification is essential.
f. "Of crucial importance in the construction of large and complex systems is a good set of tools and a suitable design method."

In contrast to VLSI design, software engineering has seen much research directed toward design methodologies, and the application of principles from these methodologies to VLSI design is a promising and relatively unexplored idea. Such adaptation of software engineering methodologies

to VLSI design is certainly nontrivial, because the latter must take into account additional concerns such as geometric and electrical limitations. Nevertheless, it is readily conceivable that transfer of concepts from software engineering could contribute to the development of an improved theoretical base in VLSI design. Such improvement would aid both the teaching of VLSI design principles and the construction of more capable automated design tools.

Despite the quantity of methodological research in software engineering, however, most modern software design methodologies do not specifically address the factor of change in the design process. In this regard, the software design techniques of D.L. Parnas are conspicuously exceptional. His techniques, specifically those of *information hiding, hierarchical structuring for design families,* and *precise specification,* have demonstrated the potential to become the nucleus of a coherent approach to the software change management problem. The overall motivation for my current research, then, has been to investigate how these techniques can be applied to manage change in the VLSI design process.

1.3 Determination of Research Objectives.

A research program that conscientiously investigates the transferability of Parnas's techniques to VLSI design change management would include:

a. An extension of information hiding, hierarchical structuring for families of designs, and precise specification into the VLSI domain by:
   (1) developing a realistic model of the contemporary VLSI design process;
   (2) identifying in the model the points at which decisions crucial to the techniques are made;
   (3) constructing for these decision points a set of criteria that guide effective design change management.

b. The development of an integrated set of VLSI tools that enable a designer to use the criteria developed in task *a* to manage design change.

c. An assessment of the effectiveness and extensibility of the design change management procedure so devised.

The scope of such a research program clearly exceeds that of a single dissertation. Consequently, it has been necessary to identify and order the component research problems embodied by this program.

1.3.1 Component Research Problems and Their Interdependencies.

1.3.1.1 Specifying Abstract Interfaces for VLSI Designs.

Britton, Parker, and Parnas [Brit81a] define an "interface" between two programs as "the set of assumptions that each programmer needs to make about the other program in order to demonstrate the correctness of his own program." An *abstract* interface specification, in their sense of the term, has been carefully limited in content to a description of *only* these assumptions, so that the specification describes not a single interface but a class of interfaces. More than one module or design thus fits the interface, and the interface is robust under certain types of changes.

Abstract interface specification of VLSI design modules is a largely unstudied problem. This lack of attention is surprising in light of the growing realization that existing VLSI design specification techniques are inadequate:

> there are no real tools for developing and simulating high-level specifications, so most work must be done by hand. There are also problems with casting the specification in the proper form....The process of specification is intimately involved with the kinds of design tools available [Wall84].

I believe that the primary reason for this inadequacy is that most references to "specification" in VLSI are to *clear-box* specifications, frequently at the logic level. Clear-box specifications address module interfaces only indirectly, by prescribing an implementation of internal module logic or circuitry. Clear-box specifications, then, blur architectural and implementational concerns; abstract interface, or "black-box," specifications keep these concerns separate. At VLSI complexity levels, separation of these concerns is essential. Clear-box specification techniques do not scale up, as can be plainly seen from the difficulty of inferring the function from even an MSI-level circuit diagram. More seriously, however, the merging of architectural and implementation issues in large-scale system design significantly hinders the partitioning of the design task and degrades (if not destroys) the conceptual unity, hence the usefulness and robustness to change, of the resulting system [Blaa81]. The development of effective abstract interface specificatio techniques for VLSI designs is thus of critical importance.

1.3.1.2 Quantifying Ease of Change of VLSI Designs.

One criticism justly raised about methodological research is that claimed benefits for proposed methods are frequently unsubstantiated. To avoid this criticism, one would like to have an impartial means of quantifying the degree to which a given VLSI design is easy or difficult to change; then, based on this quantification, the merits of various change management techniques could be compared.

Preliminary investigation into quantifying ease of change suggests that, first, a metric is needed for change itself, so that the "difference" between two VLSI designs can be measured. A modest amount of experimentation into developing a change metric was conducted in the context of this exploration [Gros84]. This experimentation investigated measuring VLSI design change as a difference in design information content, going on to attempt to measure design information by counting various discrete design components (analogous to software science approaches such as those described by Halstead [Hals77], McCabe [McCa76], or Albrecht and Gaffney [Albr83]). Techniques employing such approaches will probably need to be complex to capture design information content successfully. Further, there may be assumptions embedded in the design that render desirable information components uncountable, suggesting that such metrics must be derived from a design representation that also embodies these assumptions. An abstract interface specification for the design is such a representation.

1.3.1.3 Using Information Hiding in VLSI Design Modularization.

The decomposition of a VLSI design into modules is an important phase of the design process. In a study of decomposition criteria, Heath [Heat83] has found that information hiding [Parn72b, Brit81a] is a desirable basis for VLSI design decomposition whenever robustness under change is a primary design objective. At the same time, however, he notes that VLSI design modularization, using any criterion, requires the external aspects of each module to be "sufficiently and precisely specified." Because information hiding and modularization are so directly interconnected, precise interface specification techniques for VLSI designs are essential if the benefits of information hiding are to be obtained.

1.3.1.4 Creating Broad Families of VLSI Designs.

Parnas [Parn76a] characterizes a set of programs as a family "whenever it is worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual family members." It would be useful to learn whether considering the successive versions of an evolving VLSI design as a family has the potential to reduce the cost of VLSI design development and maintenance. There is some evidence in the affirmative [Gros83b].

Nevertheless, exploiting the family concept depends critically on the availability of precisely-specified *intermediate designs* that can serve as checkpoints for design backtracking. To the extent that these intermediate designs can be characterized by their interfaces, progress in research seems to hinge once again on the availability of techniques for VLSI design interface specification.

1.3.1.5 Determining Extent of Required Revalidation Following VLSI Design Change.

The decision to change a VLSI design brings about the following activities.

a. Determine the nature and scope of the change required.
b. Perform the change.
c. Ensure the correctness of the design by testing following the change.

Full testing of the design following each change, however, is costly and often unnecessary. Unfortunately, there currently exist no suitable ways to determine which subsets of the design might have been affected by a given change; thus one cannot be sure that anything less than full testing will suffice. Techniques are required to assist the designer in making such a determination.

This problem is a chief reason that information hiding was developed in the software domain. An extension of information hiding to VLSI design that would address this problem requires that an abstract interface be specified at the boundaries of each design component to which change effects are to be localized. The existence of such a specification would reduce testing of any changed design to the assurance that each changed module continued to meet its interface specifications. Even if such interface specifications were not met, the affected boundary modules would be clearly identified for follow-on modification and testing.

1.3.2 Dissertation Overview.

The dissertation is organized as follows. Chapter 2, after defining key terms and criteria, summarizes the research reported in the literature to date in developing methods for VLSI design interface specification. In chapter 3, I present an abstract interface specification method, an extension of Parnas's software specification techniques. I do not contend that either the specific method described, or the notation used to express it, is optimal; rather, I argue for the desirability of the approach thus exemplified.

Its advantages and disadvantages are as follows:

— The approach achieves practical scalability to VLSI design levels through precise partitioning of concerns and the exploiting of abstraction: the complexity of the specification appears to grow slowly with design scale. Pressing the separation of concerns to the ultimate, I made a major decision, to partition module semantics per-pin (instead of per-module, as is conventional). This radical partitioning enhances scalability, complexity control, and change management, at the cost of making it somewhat more difficult to specify constraints global to the module.

— The proposed specification starts by capturing high-level concerns that are then evolved into more particular specifications as details become available over the course of the design. The specification not only evolves in detail, it remains an active component of the design, subject to intentional revision occasioned by new insights from the detailing.

— The same mechanisms are used for each of functional, electrical, and geometric information.

— A new technique is included for attaching a value to the degree of specification adherence of a candidate module. Using this technique, the specifier can communicate knowledge of the marginal utility of design tradeoffs to the implementer. The technique is complex, however, and needs to be made more efficient to be practical.

The proposed interface specification method consists of:

— A black-box finite-state-machine model, together with a generic data type for pin and internal state variables (the *generic state type*) identifying the *level* of the specification. Specification refinement corresponds to evolutionary refinement of this data type. Data type syntactic and semantic components are distinguished, the former being statically verifiable and the latter requiring dynamic verification. This partitioning of concerns permits further simplification.

— An extension of Parnas's *access function* as a key specification element, partitioning module function into per-pin components.

— *Adherence functions*, which attach a value to the degree to which a module adheres to a given specification. Specification *adherence* is defined in terms of a Zadeh fuzzy set [Zade75, Zade84].

I next discuss sequencing and performance issues in the composition of such specifications. I show how access functions are treated as "communicating sequential processes" (CSP) in Hoare's sense, and show how his CSP notation can be used to express these functions. I also address four issues relating to such use of CSP:

— The power of CSP as an interface specification notation.

— Rules of thumb for expressing data and control pin access functions in CSP.

— The degree of module function partitioning that can be attained in various situations. The worst case is one in which a single master control pin effects all module function.

— Requirements of a base language for implementing CSP. CSP is a notation, not a language.

In closing chapter 3, I cite specific characteristics of the VLSI design process that might be used to simplify specifications. In general, these involve restrictions on the general model:

— Restriction to the *use of* a *standard*, *portable* base language.

— Restriction of the specification refinement hierarchy to certain predefined semantic and syntactic levels, promoting reusability of generic state type definitions.

— Restriction of the adherence function space to a set of standard adherence functions (candidates are presented).

In chapter 4, I report on an evaluation of the proposed abstract interface specification method with respect to the criteria established in chapter 2. The evaluation is based on experience with a set of four real small- to large-scale IC modules which were specified, using the proposed method, at various semantic and syntactic refinement levels. These specifications are provided in an appendix.

— To evaluate the perceived cost-effectiveness (scalability) of the method at VLSI design levels, I consider specification size as a function of design size. The data suggest that this specification approach is feasible for very-large-scale designs.

— To test the life-cycle integration of the method, I first measure the relative amount of change in the test specifications as they are refined, finding that a significant fraction of each test specification is preserved unchanged through refinement.

— To examine the spareness of the method, I informally review each of its elements for overlap with the others.

— To estimate the method's effectiveness in managing change, I conduct a series of experiments. The first demonstrates the robustness of subordinate specifications when a parent specification is changed. The second illustrates the conditions under which specification changes affect nearby modules. In the third, real-world changes are applied to an actual specification with only minor effects.

Chapter 5 contains conclusions and suggestions for future research.

CHAPTER 2

SURVEY OF PREVIOUS RESEARCH

The previous chapter suggested that effective techniques for specification of abstract VLSI design interfaces would help manage change in the VLSI design process. This chapter will describe and evaluate the effectiveness of existing VLSI design interface specification methods.

Some historical background will place this survey in context. A rough parallel can be drawn between the evolution of design methodologies for software and for integrated circuits, although the paths are offset in time by perhaps 10–15 years. In the software community, there was little recognition of the importance of specifications before system complexity exceeded manageable bounds. Even now (as Yeh [Yeh83] observes), certainly not all software designers have internalized the recognition that the complexity of modern systems is truly beyond their intellectual span of control. One expects, therefore, a parallel reluctance by IC designers to admit that complexity has become overwhelming. Because MSI and LSI complexity levels do not compel an emphasis on specification, "specification" and "interface" are terms that have yet been mentioned only infrequently in the IC design literature.

This is not to say that the specification and interface definition functions have not been performed in IC design, or even in VLSI design. These functions, however, have been called by different names, leading to some semantic confusion. To define them clearly here, I will first provide a model of the VLSI design process, a model that will show the role of VLSI design interface specifications. Based on this model, I will define "VLSI design abstract interface specification." Finally, I will survey the important published work that has been done so far in developing VLSI design interface specification techniques.

2.1 Definitions.

2.1.1 The Integrated Circuit (IC) Design Process; the VLSI Design Process.

Definitions. The *IC design process* is a sequence of elaborations that synthesize, from a design concept, a form suitable for fabrication. The *VLSI design process* is the IC design process applied to the design of very-large-scale circuits.

Discussion. Figure 2-1 illustrates a hierarchical model of the contemporary IC design process. The designer starts with a concept, either of the complete system or of a system component. The goal of the design is a description of the design in a form suitable for fabrication [Trim81]. To manage complexity, the designer subdivides the monolithic concept-to-fabrication transformation into a sequence of elaboration steps, the result of elaboration (*synthesis*) at each each step being an intermediate *representation* (or "description"; cf. [Lehm84]). Each representation, then, contains more information, in Shannon's sense [Shan49], than its predecessor; it serves as a milestone in the descent of a tree of potential designs that could be developed from the original concept. Representations frequently employed in IC design are, for example, the floorplan, the logic diagram, the layout, and so on; each embodies additional information about the design.

Synthesis and representation, however, are not the only activities that take place at each elaboration step. Two additional activities — *verification* and *evaluation* — are required at each step to ensure the accurate progress of the design effort (Figure 2-2). Redoing the current step or backtracking to a previous step may be required.

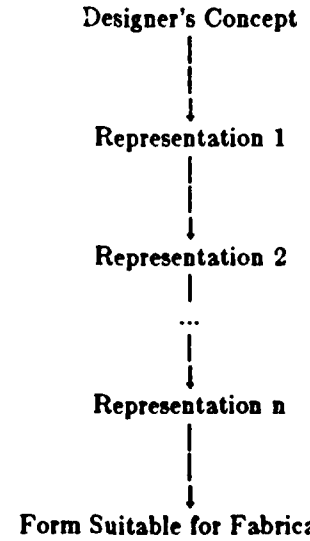


Figure 2-1. The IC Design Process.

The IC design process, therefore, is the hierarchical repetition, possibly with backtracking, of the four steps defined in Figure 2-2. Furthermore, in the IC *system* design process, Figure 2-1 should be visualized as being replicated many times, once for each design component (such as a cell), so that a representation of the total design consists of a compatible set of representations for all components.

The definition of the VLSI design process correctly identifies its component activities as the same as those for IC design. As I noted in chapter 1, however, "VLSI design is IC design in which brute force no longer works." The requirement for using complexity management techniques in the design elaborations distinguishes VLSI design from IC design at lesser levels of integration.

2.1.2 Interface; Abstract Interface.

Definitions. An *interface* (between two design modules) is the set of assumptions that each module designer needs to make about the other module to demonstrate the correctness of his/her own module. An *abstract interface* is an interface containing no assumptions about the internal composition of the module.

— *Synthesis:* the elaboration of one level of description into the next lower level.

— *Description:* the construction of an abstract model of a system (a *representation*) that details properties specific to a given purpose.

— *Verification*: a semi-formal process whereby the design at a particular level is shown to be equivalent, in either behavior or structure [or both], to one or more higher-level designs.

— *Evaluation:* the assessment of the degree to which the design meets (1) physical and performance requirements and constraints; and (2) criteria of behavioral and structural completeness.

Figure 2-2. Activities Performed at an IC Design Step.
[Boeh81, Latt81, Dall83]

Discussion. Recall the definition of "interface" given in chapter 1: "the set of assumptions that each programmer needs to make about the other program to demonstrate the correctness of his/her own program." Replacing "programmer" with "designer" and "program" with "design" yields a definition appropriate to the IC design process. Also from chapter 1, recall that an "abstract" interface specification has the additional property that the specification must not contain assumptions about the internal composition of the module (its implementation).

Nearly every representation of an IC design component is an interface specification, in that it provides the basis for other designers to draw some interface assumptions. For example, a logic diagram is frequently used as an interface specification, because it is believed that most designers can infer the interface from the diagram. But, for the reasons noted earlier, such representation-based (or "clear-box") specification techniques are inadequate at VLSI complexity levels.

Much less frequently are *abstract* interface specifications encountered in IC design. Britton, Parker, and Parnas's research [Brit81a] demonstrates that specifying an abstract interface is a deceptively difficult process: identifying and precisely specifying all the essential assumptions making up an interface, without over-constraining the definition with excess or improper assumptions, require both substantial knowledge of the project and of the underlying technologies. Nevertheless, I do not believe that VLSI designers can deal effectively with complexity and change until good abstract interface specification techniques are developed.

2.1.3 Frame; Specification.

Definitions. A *frame* is a design description assumed to be correct in all that it states or implies about the design. A *specification* is a highest-level (therefore unverifiable) description that serves as a frame for subsequent verification activities.

Discussion. Dallen [Dall83] provides a good general definition of the term *specification:* "an idealized abstraction of the system being designed." What is the role of a specification in the design process, however? Lehman, Stenning, and Turski [Lehm84] identify it, once again in the context of the software domain, when they state:

> Calculability of program correctness implies that 'correctness' is not an attribute a program may or may not possess when considered in isolation. An external frame of reference must be provided with respect to which the calculations are meaningful. This frame of reference for establishing correctness of a program is often referred to as a *specification.*

I have shortened Lehman, Stenning, and Turski's phrase "frame of reference for establishing correctness" simply to the word *frame*. A physical frame both constrains and allows freedom within its constraints; thus it is an accurate model of a specification.

Supporting this interpretation, Levene and Mullery [Leve82] identify the role of a specification from a traditional administrator's viewpoint, but still define it as a frame:

> — [A specification] should clearly indicate what the intended system is to do, including interaction with other systems, people, media, and devices, in terms that the customer or users can understand;
>
> — It should be a complete specification for the purposes of the development agency....
>
> — While it is being produced, the partial specification document can be ... referenced for use in developing other parts. This will also allow it to be maintained whenever the need for changes is recognized, either during the development of the proposed system or later during its operational life.

Parnas [Parn77a] also acknowledges the role of specification as frame when he asserts that a specification is necessary

> — To describe the problem to be solved.
> — For communication between software engineers.
> — To free the programmer from needing to know how the rest of the system works.
> — To support the development of multi-version software.
> — To complete the description of intermediate design decisions [encapsulated, for example, in the abstract primitives that make up high-level representations in Figure 2-1].
> — To permit verification of intermediate design decisions.

A *specification*, then, is that abstract description of a system that provides a frame for system verification, for answering the question "Did we build the product right?" [Boeh81]

Since, as I have noted, the word "specification" is infrequently used in a VLSI design context, what part of the VLSI design model is, by virtue of addressing these purposes, a *de facto* specification? Consider the initial (top-level) steps in the model of VLSI design developed in section 2.1.1. Figure 2-3 expands these initial steps into their four component activities. One of these activities must be the synthesis, from the (undescribed) designer's concept, of a highest-level description of the design, a description that cannot be verified since there is no description against which it can be verified. It is this highest-level, unverified description that I call the specification of the VLSI design, and it is this description that serves as a frame for verification activities.[1]

---

[1] This is not to say that lower-level descriptions cannot be taken out of their context in one design hierarchy (Figure 2-1) and used as specifications in another. For example, a logic diagram might not be a specification in the context of a complete design but might serve as one in a subset of that design activity (a separate instantiation of Figure 2-1).

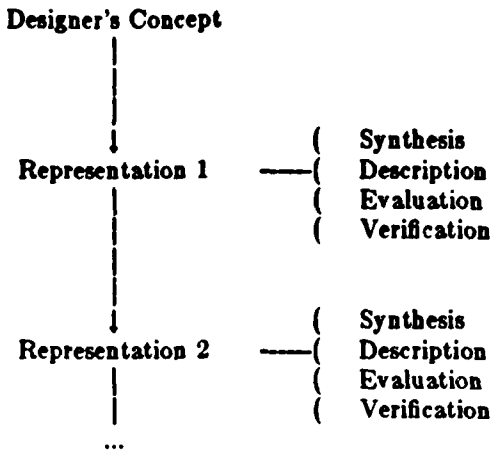


Figure 2-3. Expansion of the VLSI Design Process
at its Highest Levels.

Many different types of specification exist. The following list describes some distinguishing characteristics important for VLSI designs.

(1) Informal vs. Formal. A specification is usually called *formal* if both the syntax and semantics of its primitives are mathematically precise, that is, strictly and unambiguously defined [Lisk79]. Formality and precision in specifications are, therefore, intertwined. Parnas [Parn77a] concedes that a specification can be precise without being formal but notes that creating one is "very difficult." By definition, the converse is not true: a formal specification is always precise.

Sometimes added is the stipulation that a formal specification must contain the capability to demonstrate rigorously the validity of implementation objects derived from it [Cohe83, Lehm84]. In my view, this overconstrains the definition, however desirable such a capability may be.

(2) Syntactic vs. Semantic. Wegner [Wegn84] differentiates syntactic and semantic interface specifications of software components as follows: "Syntactic interfaces specify compile-time invariants that determine how components fit together, while semantic interfaces specify execution-time invariants that determine what the component computes." A similar concept exists for IC modules: the syntax (well-formedness) of a composition of modules can be checked statically (i.e., without recourse to dynamic analysis or simulation), whereas the semantic integrity of a module

composition requires dynamic verification. Syntactic specifications, therefore, are adequate to determine the consistency (but not the validity) of a proposed module interconnection. As Wegner notes,

> strong semantic interface specifications are intractable in the sense that they do not always exist and their correctness cannot always be verified. ... The trade-offs between the flexibility and efficiency of very weak interface specifications and the guaranteed integrity of strong interface specifications need to be better understood.

(3) Non-executable vs. Executable. A specification is called *executable* if it is possible to create a machine interpretation of the specification that provides an approximation of external system behavior [Zave84]. Because of their precise syntax and semantics, all formal specifications can be meaningfully processed by a computer [Lisk79]; however, not all formal specifications contain the kinds of behavioral semantics necessary to permit simulation of system behavior by machine processing.

An extreme (and sometimes overconstraining) form of executable specification is a realization of the desired design: when questions of specification details arise, one "asks the machine" by making an experimental query of the existing realization [Broo75].

(4) Black-Box vs. Clear-Box. Earlier, a *clear-box* (sometimes called *structural*) specification of a module interface was characterized as one that prescribed the interface indirectly by describing an implementation of internal module structure, logic, or circuitry. A *black-box* (or *behavioral*) specification, on the other hand, prescribes a module interface through a description of the module's externally visible behavior [Zave84]. Somewhat generally, I referred to a black-box interface specification as an *abstract interface specification*, in that for a given black-box specification more than one implementation (of the type prescribed in a clear-box specification) could exist.

This distinction is perhaps too clearly drawn. Proponents of clear-box specification might state that the implementation they produce is not meant to constrain the implementation, but merely to prescribe its behavior. Indeed, this implementation itself could be abstract, describing module behavior executably but in terms of implementation-independent substructures [Zave84]. Zave calls a specification that describes module behavior in such a way an *operational* specification. In contrast, a *functional* or *input/output* specification describes a direct functional relationship among module inputs and outputs without reference to substructures [Lisk79]. For this dissertation, an (abstract) operational specification is also a black-box specification, but a so-called operational specification that prescribes an implementation is neither black-box nor operational.

In this dissertation, I will thus not join the debate between proponents of operational and functional specifications. The crucial issue is that both types can specify interfaces abstractly, and I contend that such specification, which I call *black-box*, is superior to its clear-box counterpart. Furthermore, in the hardware domain much of what passes for operational specification, as Zave defines it, is in practice clear-box specification.

2.1.4 Module Function; Module Performance.

Definitions. The *function* of an IC module is its externally visible behavior. The *performance* of an IC module is a set of measurements of its function in terms of physical phenomena (e.g. speed, power, area).

Discussion. The word "functional" is overloaded in specification parlance. In distinction from the use made of it in the previous section as an opposite to "operational," it can also be used to differentiate the semantics contained in the specification.

In software, Liskov and Berzins [Lisk79] state that "functional" specifications "describe the effect of the module on its external environment," whereas "performance" specifications "describe constraints on the speed and resource utilization of the module." In IC design, however, the effect of the module on its external environment *depends* on its performance, so that the line between functional and performance specifications must be redrawn. Indeed, it is possible to think of a single function/performance continuum, corresponding to a continuum of abstraction of specification primitives. I will treat this issue in detail during the discussion of adequacy in section 2.2.2.

Other-than-functional specifications are sometimes called *constraints*. Roman [Roma85], in a recent taxonomy of software requirements engineering issues, enumerates several constraints besides performance: interface constraints (assumptions about the environment), operating constraints (size, weight, power, etc.), life-cycle constraints (maintainability, enhanceability, portability, development time limitations, etc.), economic constraints, and political constraints. For the purposes of the current VLSI-oriented work, Roman's interface constraints are considered to be orthogonal to the functional/performance issue; operating constraints are considered together with performance constraints; and the other three types of constraints can be dealt with only informally.

2.1.5 Abstract VLSI Interface Specification.

Definition. An *abstract VLSI interface specification* is a reduced set of assumptions that a designer of an adjacent VLSI module would need to make to design, and especially to verify the correctness of, this adjacent module. This set is reduced in that it excludes assumptions that have to do with internal module implementation.

2.2 Requirements for VLSI Design Abstract Interface Specifications.

In this section, I shall enumerate the requirements that an abstract interface specification must meet in the VLSI design process. Later, I shall examine existing specification methods in light of these requirements.

2.2.1 Provision of Frame for Verification (Adequacy).

This requirement follows directly from the definition of "specification" developed in section 2.1.3. A description purporting to be a specification must *be* a specification: it must provide a frame for both technical and administrative verification. (To the extent that these types of verification differ, technical verification focuses on technical performance of the product, while administrative verification examines the product for fulfillment of relevant contractual obligations.)

Design specifications can be constructed at differing degrees of what I have chosen to call *adequacy*. That is, there exists a continuum (Figure 2-4) of refinement of the amount of detail included in the specification. At the coarsest levels, a specification may be given in general terms, e.g. "this module is the ALU." Refinement of this specification along the continuum of Figure 2-4 gradually adds detail to the specification, until at some point in the continuum there is an adequate level of detail to guide the implementation. At this point, which is admittedly vaguely defined, it can be said that the specification is *adequate*.

Liskov and Berzins [Lisk79] distinguish two types of specifications: *functional* and *performance*. With respect to these two types of specification, the experiences of software and hardware engineering differ. In software engineering, functional specifications are of chief importance, because performance issues are largely orthogonal to function and thus can be abstracted away; if better performance is needed, one can tune the implementation later or get a faster machine.

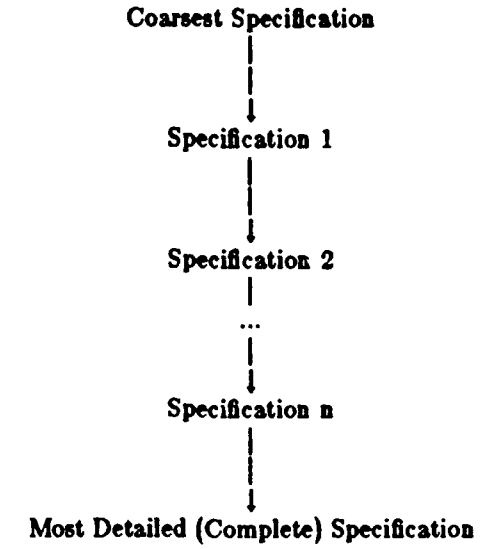


Figure 2-4. Continuum of Specification Refinement.

Two separate continua of refinement (Figure 2-4), one each for function and performance, thus exist in software specification.

In hardware engineering, however, a well-known adage states: "Circuits must not only work; they must *perform*." In hardware design, what is functionally correct under less specific parametric or performance constraints may cease to be functionally correct under more specific constraints; thus function and performance are naturally intertwined. Because of this characteristic of hardware engineering, then, in IC design what is called a functional specification can be interpreted as an early stage, and performance specification as a later stage, of a single specification refinement continuum. At all levels of refinement the specification must be *precise*; the primitives with which it is described, however, may vary in specificity. The adequacy, then, of an IC design abstract interface specification depends on the specificity of its performance constraints relative to the specificity required. If one's concern is merely that the circuit function correctly at some level of performance, then a less detailed (functional) specification will be adequate. But if one's concern is that the circuit function correctly at an intended level of performance, a more refined, more detailed specification will be needed.

In VLSI design in particular, concern for correct behavior at intended performance levels is of central importance. Ultimately, then, a VLSI design interface specification must contain performance detail. This requirement implies, because of the inseparability of performance from a design's geometric and electrical characteristics, that these latter characteristics must also be included. As a result, a substantial variety of information is required in a VLSI design abstract interface specification, and the specification technique employed should both (1) provide rich enough facilities to express this information and (2) express it in terms of the performance levels required.

Shaw [Shaw84] has recently elaborated on this requirement as follows:

> A specification methodology that addresses [properties other than pure functional correctness] must have two important characteristics. First, it must be possible for the programmer to make and verify assertions about the properties rather than simply analyzing the program text to derive exact values or complete specifications. This is analogous to our approach to functional specifications — we don't attempt to formally derive the mathematical function defined by a program; rather, we specify certain properties of the computation that are important and must be preserved. Further, it is important that the specification methodology avoid adding a new conceptual framework for each new class of properties. This implies that mechanisms for dealing with new programs should be compatible with the mechanisms already used for functional correctness.

2.2.2 Spareness.

Parsimony, that is, limiting the number of related expressive mechanisms in a specification technique, is clearly desirable. It might fruitfully be thought of as an aspect of a more general economy, which I shall call *spareness*. Spareness is the property of a specification that implies that it says only what it needs to say.

As Parnas has noted [Parn72a, Parn77a], what a specification does not contain is nearly as important as what it does contain. It is vitally important not to clutter the specification with extraneous requirements: these not only over-constrain the implementer but also slow comprehension [Meye85]. Thus a spare abstract interface specification technique must not only manifest economy of expression, but it must also identify which details are essential and which are unnecessary.

2.2.3 Perceived Cost-Effectiveness.

The desirability of spareness is but one illustration that the usefulness of a specification is strongly related to its intuitive appeal to designers, that is, the degree to which the specification technique is *perceived* as cost-effective. Clarity and simplicity, as Liskov and Berzins [Lisk79]

have pointed out, contribute significantly to a specification technique's attractiveness. Choosing familiar representations also aids acceptance [Lisk79]. Finally, the costs of using the method need to be reduced as much as feasible: a clumsy or slow user interface is certain to undercut adoption of a proposed technique, especially considering that designers have not yet embraced specification methods *per se*. Baroque and unusable specification methods are all too easy to develop: Liskov and Zilles [Lisk75] called difficulty in use "the fundamental problem with specifications."

2.2.4 Malleability.

A strong contributing factor in the perceived cost-effectiveness of a specification is its usefulness *throughout the design life-cycle*. What quality of a specification contributes to such usefulness?

A valuable but elusive goal in design is that of getting the specification "correct" the first time through careful axiomatic thinking. Several noted contemporary scientists, such as Dijkstra, believe that such a goal is achievable [Berg81], and they may ultimately be proven right. For the present, however, specifiers do not always seem to be able to foresee the later emergence of difficulties and even contradictions in the specification constraints they have established at the outset. If these specification constraints are rigid, backtracking that includes constraint modification would be impossible.

Perhaps, then, it is unfruitful to visualize specification constraints as being rigid. Instead, the constraints might better be viewed as variables, albeit with "stiff" coefficients of elasticity. (Other variables have more pliable coefficients.) I call the degree to which a specification's constraints can be viewed as variables its *malleability*, and I contend that malleability enhances the life-cycle utility of a specification. Malleability is not imprecision; it *is* the property of a specification that permits and facilitates precise constraint adjustment as necessary, in the interest of preserving the usefulness of the specification as a frame for verification throughout the design life-cycle.

Good examples of malleable specifications are the incremental (prototype-based) [Myer84] and transformation-based [Balz83, Part83] life-cycle paradigms, as contrasted to the conventional linear development model. Such methods depend on regular incorporation of feedback into the specification, which is then used directly to derive refined systems. The appeal of these non-conventional approaches may indeed be due largely to their increasing the usefulness of the

system specification in the design life-cycle.

2.2.5 Change Management Support.

Finally, as has been noted in chapter 1, a primary motivation for developing a new VLSI design interface specification method is to support change management. To do this, an ideal specification technique should assure the designer that internal changes to a design module that do not change the specification also do not have effects outside the boundaries of that module. This requirement thus overlaps the requirements of the preceding sections, especially those for adequacy and malleability.

2.3 Existing VLSI Design Interface Specification Methods.

No technique fully meeting the five requirements enumerated in the preceding section has yet been developed. At this point, I will review the important published work that has been done in developing methods for constructing VLSI design interface specifications. It is important to note that all the specification methods to be surveyed have a place in the design process; however, I shall evaluate them, using the criteria of section 2.2, specifically as candidates for interface specification methods to support change management.

2.3.1 Methods for Informal Specification.

Informal specifications of IC interfaces are most often an *ad hoc* mixture of prose descriptions and block diagrams, intended to convey the function and/or structure of the design to a general audience. Procedures for informal specification rarely receive wide dissemination.

2.3.1.1 Examples.

— Lattin *et al.* [Latt81] mention a "200-page document" that served as the specification for the Intel iAPX-432 design and that, judging from the context, was probably at least partly informal. Indeed, it is reasonable to surmise that most major chip design projects begin with such a document.

— Standard cell system documentation usually includes an informal description of cell interfaces. Module interfaces in the Stanford Cell Library [Newk83], for example, are described both in prose and in a formal mask description that can be simulated to provide more precise interface data. Informal specifications similarly are included in documentation of those silicon compiler systems [Gros83a] that are based on parameterized standard cells, such as the Concorde System of Seattle Silicon Technology, Inc. [SST84].

2.3.1.2 Strengths/Weaknesses.

Britton, Parker, and Parnas [Brit81a] identify the strength of informal specifications: they are easy to review for people not intimately familiar with the technical details of the project. Since most specifications require approval by persons in this category, Britton, Parker, and Parnas suggest that software specification methods should accommodate an informal component that embodies those assumptions critical to the design and that is consistent with the formal portion of the specification.

Unless they are supplemented by a formal component, however, informal specifications are inadequate to provide a verification frame in the design of complex systems, for the reasons listed by Dasgupta [Dasg84]:

> — The dynamic aspects of the architecture ... are usually specified incompletely. [This is particularly true in IC module interface specification, since the interface is semantically rich, with functional, electrical, and geometrical components, and an informal specification is usually truncated for manageability before all these data are included.]
>
> — It is extremely difficult, even in principle, for the design to be validated for correctness without constructing and testing the physical system.
>
> — The sequence of design decisions and the rationale for them are seldom documented explicitly. ... It is virtually impossible to investigate, manipulate, or alter a design, or evaluate alternative architectures for performance characteristics, without constructing and testing the physical system. [Heninger [Heni78] blames this lack of robustness not only on the intractability of informal specifications but also on their frequent failure to include fundamental assumptions, required subsets, expected changes, and rules for treatment of undesired events.]

The inherent weaknesses of informal specifications as stand-alone system specifications are discussed in greater length by Meyer [Meye85].

2.3.2 Methods for Syntactic Specification.

Syntactic interface specifications exist for verifying the integrity of a composition of modules; they are of most value when such verification is complex.

2.3.2.1 Examples.

The best examples of syntactic IC module interface specification are "pin-typing" schemes. Ségal [Séga91] was among the first to demonstrate such a technique, but his effort was abandoned because of incompatibility with other existing tools. Noice, Mathews, and Newkirk [Noic82] reported on the benefits of using signal-labeling conventions in the two-phase nMOS clocking discipline described by Mead and Conway [Mead80]; Karplus [Karp84] extended their work and provided a formal basis for it. Some success with a related scheme using a different set of restrictions has been reported by Poulton [Poul84]. Although he has so far used only manual verification of the embedded information, his method is compatible with existing design tools and has been reported to aid design composition significantly in more than one project involving multiple clocks [Poul85]. Newkirk and Mathews [Newk83] and the designers of the VIVID system [Rose84] have provided some syntactic pin-typing facilities in their respective design tools; no results analyzing their usefulness have yet been reported.

2.3.2.2 Strengths/Weaknesses.

In the previously-cited quote, Wegner [Wegn84] stated correctly that although syntactic interface specifications are easier to create, it is usually in verifying semantics that a verification frame (a specification) is most required. Consequently, purely syntactic IC interface specification techniques have not yet generated much interest, except in designs, such as Poulton's, of significant syntactic complexity.

2.3.3 Methods for Non-Executable Specifications.

Because informal interface specifications have already been considered, I shall examine only formal, non-executable specifications here. There have been two distinct formal, non-executable approaches, which now appear to be joining in a confluence of thought. The first approach provides a formal high-level description of both behavior and structure as a starting point for design synthesis. The second, traditionally less concerned with the specification's role in aiding synthesis, primarily describes module behavior or function with a view toward providing a frame in a formal proof of design correctness.

Synthesis-directed specification approaches begin with formal descriptions written in documentation languages that normally capture both behavior and structure of the intended design. Many of these languages are called Computer Hardware Description Languages (CHDLs). While interpreters have been created to render some of these CHDLs executable, in the main their purpose has been documentary, and their role in the specification process has been as an informal point of reference during design elaboration.

There is a class of synthesis-directed languages in which the role of capturing a starting point for synthesis so far outweighs the specification role that the languages are rarely referred to as CHDLs. These might be called "generator source languages," because their role is analogous to that of software source languages. Descriptions written in such languages (1) usually constitute an intermediate step between a (perhaps informal) specification and lower-level synthesized descriptions, and (2) serve as input to generators that produce executable design descriptions. Here are included primarily input languages for automated synthesis aids, such as chip assemblers [Mudg81, Katz83, Katz84] and silicon compilers [Gray82, Ance83, Lipt83, SST84].

Proof-directed specification approaches have undergone a change of direction. They began with non-executable specifications, such as the non-procedural functional language used by Wagner in 1977 and cited by Barrow [Barr84]; such specifications were reconciled with synthesized designs using manual proof techniques. Lately, however, with the renewed interest in artificial intelligence, more interest has been shown in creating proof-directed specifications that can be executed by inference engines to prove design correctness automatically; thus most contemporary proof-directed specifications are executable.

A natural question, raised by Cohen at the 1983 International Conference on VLSI [Cohe83], is whether a single interface specification technique could serve both synthesis and verification needs. That is, can a single technique capture sufficiently the behavior and structure of the desired design to guide the synthesis effort, while at the same time being precise enough to support formal proof of design correctness?

2.3.3.1 Examples.

Since Dallen [Dall83], Dasgupta [Dasg84], and Nash [Nash84] provide excellent surveys of CHDLs, it suffices to note here that CHDLs can be grouped into three classes, the block languages (e.g. PMS [Siew82]), the register-transfer (R-T) languages (e.g., CDL [Chu74], DDL [Diet74,

Maru85], ISP [Siew82], ISPS [Barb82]), and the graphical languages (e.g., Petri nets [Pete77], Patois [Dall83], Interface-Nets [Moln85]).

CHDLs have proliferated furiously [Wern84]. One current trend is toward a consensus CHDL; an international group has been working for more than ten years on such a language, CONLAN [Pilo85]. Another current trend seems to be toward extensible omnibus CHDLs that can be used to describe in a single language function, structure, and layout (e.g., the VHSIC Hardware Description Language (VHDL) currently under development by Intermetrics for the U.S. Department of Defense [Dewe84, Shah85] and GTE Laboratories' Zeus [Lieb85]). Finally, fine results have been obtained using programming languages as CHDLs, such as the use by Blaauw of APL for this purpose [Blaa76, Blaa83].

Barrow's VERIFY system [Barr84], although not properly a member of this section because it produces executable specifications, typifies the state of the art in proof-directed specification approaches. He begins with a set of primitive modules, represented as finite-state machines with known black-box function. Next, he composes these modules into the structure of the target design. Finally, he uses a PROLOG-based interpreter to prove that the mathematical composition of the functions making up the interconnection is the function of the target design.

Can a single specification support both synthesis and verification? Frankel and Smoliar [Fran79], Rowson [Rows80], Gordon [Gord81], Cardelli and Plotkin [Card81], Hafer and Parker [Hafe83], and Sheeran [Shee84] have all suggested the feasibility of specifications that apply mathematical compositional systems to VLSI design components. Subrahmanyam [Subr83] has recently reported on the development of a theoretically precise specification language designed to serve as input to a silicon compiler, thereby showing *a fortiori* that it can serve to guide manual design synthesis.

2.3.3.2 Strengths/Weaknesses.

Formal specification methods have substantial potential for dealing with the complexity levels inherent in VLSI design, in that they can capitalize on their foundations in discrete mathematics and the theory of computation. Nevertheless, the formalism in such methods is now largely foreign to programmers and managers, even to those with engineering backgrounds. If these methods could be interpreted in automated tools, the retraining necessitated by their introduction would be lessened, although certainly not eliminated. But methods not optimized for the use of

such tools, such as non-executable formal specifications (which now usually require a manual interface), will be slow to be accepted.

On the other hand, methods that are familiar, such as clear-box specification using CHDLs, will be abandoned reluctantly, and the use of such techniques is currently widespread [Dasg84, Evan85]. CHDLs for black-box specification, whose primitives are abstract, do not have the same intuitive appeal, however. Because no CHDL has emerged as a standard after many years [Wern84], a needed breakthrough in CHDL technology may still be lacking.

2.3.4 Methods for Clear-Box Specification.

Since non-executable specifications were addressed in the preceding section, only executable clear-box specifications will be covered here.

Clear-box VLSI specification methods are an evolution of specification techniques used at lesser degrees of integration. The primary characteristic of such clear-box specifications is that they prescribe interfaces by inference, rather than directly.

2.3.4.1 Examples.

Executable clear-box specifications can be classified according to the level of abstraction of the primitives appearing in the clear box. The executable clear-box CHDL specifications have the highest-level primitives, such as registers. Next, generally for smaller modules, are found the ubiquitous logic diagrams and their simulators. These are especially prevalent in the composition of systems consisting of SSI and MSI circuits. Finally, in cases in which a replacement for an existing module is being developed, circuit- and mask-level simulators are available to enable the existing module to act as the specification. Each technique is widely used, and there are many languages and simulators for this purpose.

Excellent results have been reported from the use of a clear-box specification system by Dussault, Liaw, and Tong at AT&T Bell Laboratories [Murp83, Duss84]. This tool, called the Functional Design System (FDS), enables designers to construct circuits hierarchically using custom-designed primitives tailored from a menu of options. Designers can work with black-box descriptions of each primitive written in a special executable language. FDS is currently being used in a production environment.

2.3.4.2 Strengths/Weaknesses.

Clear-box specifications, like informal specifications, have the advantage of being familiar to designers. They also provide a path to a feasible implementation, whereas black-box specifications can mask fatal implementation difficulties [Moln85]. From a VLSI design point of view, however, clear-box specifications also have several disadvantages, as section 1.3.1.1 states. The most serious is that they confuse architecture (the external view) with implementation, elevating low-level concerns too early in the design process to achieve good design partitioning and unity. Furthermore, as Parnas notes [Parn77a], it becomes "very hard for both the reader and writer of specifications to distinguish requirements from peculiarities of the sample implementation." Thus clear-box specifications may constrain the implementation unnecessarily.

Zave [Zave84] lists further difficulties. Clear-box specifications are complex; often not taking advantage of abstraction, they scale up poorly and are hence reduced in value at VLSI levels of integration. Furthermore, because of this level of detail, they are time-consuming to construct and analyze. Finally, Dallen [Dall83] contends that clear-box specifications do not naturally aid the synthesis process in hierarchical design because they are usually tied to fixed levels of abstraction.

2.3.5 Methods for Formal, Semantic, Executable, Black-Box Specification.

Because critical disadvantages exist with each method of VLSI design interface specification discussed thus far, examining the intersection of their complements provides the best hope for finding techniques meeting the five requirements of section 2.2.

Functional specification using high-level programming languages or special-purpose specification languages is the primary technique that appears in this intersection. Typically, a black-box interface specification is constructed for the module under design, a high-level language is used to describe its behavior, and the simulated output is compared to that obtained by simulating lower-level representations of the design. Evaluation of the specification through simulation, rather than through analysis or prototyping, is almost always used because of the intractable complexity of the specification at VLSI levels of integration. Special-purpose specification languages have arisen because programming languages can be cumbersome to use for this purpose: they do not handle timing in a natural way, and it is difficult to make specifications written in them grow with the design life-cycle.

2.3.5.1 Examples.

Many designers use languages such as APL, Algol, BLISS, C, and Pascal for functional simulation of system specifications [Dall83, Evan85]. Lattin *et al.* [Latt81] used a high-level language, SIMULA, *directly* as the specification medium for the Intel iAPX-432 chip. That is, rather than serving as a simulator for a separate specification, this simulator itself filled the role of the frame for design verification; questions were resolved by "asking the machine." Tham, Willoner, and Wimp [Tham84] of Intel updated this technique in their recent use of a MAINSAIL frame; a similar approach was also used by the designers of the Hewlett-Packard FOCUS chip [Cane83]. The MetaLogic, Inc., MetaSyn silicon compiler [Sisk82] accepts as input a LISP-like executable description of the function to be implemented [Wall84]. Control Data Corporation's MIDAS system specifies designs using a programming language enhanced with timing semantics [Evan85]. Finally, Suzuki has used Concurrent Prolog to specify the function of the Dorado computer [Suzu85].

Parker and Wallace [Park81] identify the following milestones in the history of special-purpose black-box interface specification languages:

— Bell and Newell's port semantics [Bell71];
— Curtis's Interface Description Language [Curtis], a multi-level extension of PMS and ISP developed circa 1974-1975;
— Vissers's formal description[2] of state diagrams using APL enhanced with timing constructs [Viss76]; and
— Marino's proposed MPLID [Mari78], a language for specification of hierarchical module interfaces.

Parker and Wallace's own language, SLIDE (Structured Language for Interface Description and Evaluation), synthesizes these developments into an executable black-box register-transfer language. SLIDE's major contributions are this synthesis of trends and its elegant facility for describing module interconnections as communicating asynchronous concurrent processes.

[2] See also [Blaa76]

	Pin names/attributes	Parameters
T1	(K/unspecified (L/input (M/unspecified	shape=6 lineage=pure-enh.
T2	(N/unspecified (P/input (Q/unspecified	shape=3 lineage=pure-enh.
T1/T2 composed in parallel	(K/unspecified (L/input (P/input (M/unspecified	shape=2 lineage=pure-enh.

Figure 2-5. Bain's External Outlines [Bain84].

Several other interesting approaches have recently been taken to the development of executable special-purpose languages for black-box VLSI design interface specification. For example, Bain [Bain84] has incorporated into his CHECK-ME design methodology verification system a technique, based on Penfield's [Penf72] "wiring operators," which completely characterizes a circuit using an "external outline" consisting of a list of terminal names, a limited set of input/output attributes, an electrical drive factor based on the shape of the internal transistors, and a broad characterization of the implementation strategy that he calls "lineage" (Figure 2-5). While it requires extension to achieve semantic adequacy at greater levels of specification refinement, Bain's method is encouraging, for it illustrates the simplicity that can be achieved in abstract IC circuit interface specifications if an appropriate set of abstractions is identified.

Tsai and Achugbue [Tsai83] provide in their BURLAP system an idea of what a timing-level extension of Bain's external outline might look like. They specify the abstract interface by a "multi-cell module" (MCM) description consisting of the components listed in Figure 2-6. Figure 2-7 shows the contents of a typical BURLAP specification.

The hierarchy exploited by Bain and by Tsai and Achugbue has been formally analyzed by Chen and Mead [Chen83]. Using mathematical methods, they define a *semantic hierarchy* of specification, and of the ensuing verification by simulation, similar to that depicted in Figure 2-4. In this way, they formally partition the design process and suggest that a uniform representation can be used to provide interface descriptions as simulator input at all levels. Individual design

I/O Interface	–	indicates the pin names and the pin types at the "black box" level.
Behavioral	–	description of the block function, either in a procedure or Boolean equations (truth table).
Structural	–	description of the interconnection of the low-level primitive elements into which the MCM can be expanded [not used in black-box specification].
Physical	–	description of the layout and size of the MCM.
Electrical	–	delay and timing information used by logic simulator and timing verifier.

Figure 2-6. Components of a BURLAP Interface Specification [Tsai83].

```
Begin_Cell
    Name:                RAM32x8
    Class:               Storage_Element
  Physical
    Cell_Width:          40
    Cell_Height:         40
    Pin_Count:           23
  Electrical
    Cell_Current:        35mA
    Power_Dissipation:   150mW
    Delay:               60ns
  Pins
    Pin_Name:            A0
      Pin_Type:          IN
      Capacitance:       0.1pf
      Equivalent_Port:   0,38
      Layer:             Polysilicon
    Pin_Name:            0
      Pin_Type:          OUT
      Capacitance:       0.15pf
      Equivalent_Port:   40,38
      Layer:             Polysilicon
    [Pin_Name:           ...]
  Power-Bus/Clock
    VSS_Width:           3
    VSS_Port:            38,0
End_Cell
```

Figure 2-7. A BURLAP Specification of a Static RAM [Tsai83].

modules, in their work, are conceived as separately-compilable MAINSAIL descriptions of finite-state machines, encapsulating design details at appropriate levels of semantic abstraction. To

date, however, their work has centered on creating the formal bases necessary to prove correctness, rather than on developing an abstract interface specification technique tractable at VLSI levels of integration.

Another special-purpose specification language, previously cited, that makes use of separately-compilable modules that represent finite-state machines is included in Barrow's VERIFY system [Barr84]. VERIFY also uses the hierarchical approach recommended by Chen and Mead to predict the functional behavior of an interconnection of these modules from a description of the behavior of each. But, instead of using the simulation customary in the verification process, VERIFY uses a PROLOG-based interpreter to compare the predicted behavior with a set of equations that specify the system's behavior. To date, VERIFY has been tested on functional specifications only, but its approach to verification is encouraging, in that it can be replicated hierarchically to scale up to VLSI design levels without the need for specifications to be re-executed at each level of refinement. A sample VERIFY specification is shown in Figure 2-8.

The inherent complexity of VLSI-level system specifications has hindered not only verification but also the development of silicon compilers, the hardware design counterpart of the transformational implementation of operational specifications mentioned by Zave [Zave84]. Subrahmanyam [Subr83] has addressed both problems simultaneously in his specification formalism called BehaviorExpressions, an algebraic specification language designed also to serve as input to a high-level silicon compiler. Figure 2-9 gives an example of a specification expressed in this language.

Subrahmanyam's work addresses the need for semantic adequacy at each of the functional, geometric, and electrical levels mentioned earlier; it also includes paradigms for instructing the silicon compiler to consider performance and cost data in performing its transformations. His research plan is comprehensive and ambitious, and its results could prove significant indeed.

2.3.5.2 Strengths/Weaknesses.

Functional interface specification with abstract high-level languages has achieved the most promising results of any technique discussed. Specification refinement, however, requires the eventual representation of dense semantics (particularly geometric and electrical information, concurrency, and parallelism). Although it is possible to express such semantics and their complex data types in conventional programming languages, not many designers are now familiar with the

```
% Definition of module type Register
Module: reg
Ports:
  in      (input, integer)
  out     (output, integer)
State Variables:
  contents (integer)
Output Equations:
  out := contents
State Equations:
  contents := in
```

Figure 2-8. A Module Specification in VERIFY [Barr84].

```
StackChip(s) =
                    INITIALIZE. StackChip (NEWSTACK)
                    +
                    INSERT x. StackChip (PUSH(s,x))
                    +
                    DELETE TOP(s). StackChip (POP(s));
```

where Stack is an abstract data type with predefined syntax and semantics; INITIALIZE, INSERT, and DELETE are global ports (two input and one output, respectively); s is a parameter that represents an instance of the type Stack; and NEWSTACK, PUSH, and POP are operations defined for the type Stack.

Figure 2-9. Specification Using BehaviorExpressions [Subr83].

disciplined programming style that such expression requires. Consequently, these languages are now more frequently used for functional specification alone.

Special-purpose languages possess more powerful primitives for expressing design semantics, but their degrees of perceived cost-effectiveness and malleability vary, particularly in the representation of very-large-scale designs. And, as Evanczuk notes, a special-purpose language has a smaller user community; he believes that the user momentum, availability, and portability of general-purpose programming languages makes techniques based on them more likely to succeed [Evan85].

2.4 Parnas's Techniques for Abstract Interface Specification.

Parnas's approach to attacking the complexity issues of the "software crisis" has been one of "divide and conquer." Few others have elucidated as well as he the issues involved in dividing the software design task. A brief review of his treatment of these issues is in order.

2.4.1 State Machines.

In an early paper [Parn72a], Parnas set out the *module* as his software building block and described how it was to be specified. His was a black-box specification, and his model of the black-box contents a state machine. His choice of this formal but familiar model is typical of his philosophy: understanding is the goal, and understanding is aided by simplification, without compromising precision, wherever possible.

Interaction with Parnas's module takes place entirely through what he later called *access functions* (or access programs). These functions make up a major part of a module's specification and can be invoked by other modules either: (1) (*effect access function*) to change the state of the module by providing an input value; or (2) (*value access function*) to read out the state of the module. By restricting access functions to one of these two types, the opportunity for prescription of an intended implementation (which would cause the specification to become clear-box) is reduced [Bart77].

2.4.2 Traces.

Parnas believed strongly [Parn72a] that specifications should be *minimized*, in the sense that only the information required for the specification should be provided "and nothing more." He felt that a chief source of specification failure was overprescription or redundancy, a view that was then controversial but that has now been generally accepted [Meye85]. With Bartussek [Bart77], Parnas became concerned that functions in the specification could potentially prescribe an implementation. To be sure, the definitions of such functions were, by fiat, unavailable outside the module being specified (leading to their being called "hidden functions" in the literature). But their existence bothered Bartussek and Parnas, especially because of the suggestion (see section 2.1.3) that modules could be specified "by giving a program whose behavior would be acceptable and asking that the program produced be 'equivalent'." Such "equivalent functions," they felt, provided superfluous information [Parn75a].

Syntax:

PUSH:	<integer> ×	<stack> → <stack>
POP:		<stack> → <stack>
TOP:		<stack> → <integer>
DEPTH:		<stack> → <integer>

Semantics:

A. Legality:
(1) &(T) => &(T.PUSH(a))
(2) &(T.TOP) = &(T.POP)

B. Equivalence:
(3) T.DEPTH ≡ T
(4) T.PUSH(a).POP ≡ T
(5) &(T.TOP) => T.TOP ≡ T

C. Values:
(6) &(T) => V(T.PUSH(a).TOP) = a
(7) &(T) => V(T.PUSH(a).DEPTH) = 1 + V(T.DEPTH)
(8) V(DEPTH) = 0

Figure 2-10. Specification Using Traces of a Stack for Integer Values [Bart77].

Notation:

Dot (.) is the functional composition operator.

&(T) is TRUE if calling the functions in the sequence specified in the trace with the arguments given in the trace when the module is in its initial state will not result in an abnormal exit.

V(T) is a value access function that returns information about the module state.

The result of Bartussek's and Parnas's work was an "axiomatic" specification approach called "traces." It is called axiomatic [Lisk79] because a specification using traces makes statements (axioms) only about the effects of function calls; it contains no reference to any internal data structure. Figure 2-10 gives an example of such a specification. Specifications using the trace approach are termed *complete* (i.e., they completely determine externally visible module behavior) if one can determine from the specification the value returned by every trace (sequence of function calls) ending with a value access function and not causing an abnormal exit. The use of traces for IC specification has been reported by Rem, van de Snepscheut, and Udding [Rem83].

2.4.3 The Software Cost Reduction (SCR) Project.

Parnas, sensitive to the frequently-made criticism (see section 1.3.1.2) that claims made for new software engineering methods are usually poorly supported, undertook a major project beginning in the late 1970's to test his and other software engineering principles in a real-world

environment. This project, sponsored by the Office of Naval Research, is structured around the re-engineering of obsolete avionics software for the Navy A-7 aircraft. Thus it provides a point of comparison for the efficacy of the new software engineering methods.

Fn_name	*Parm_type*	*Parm_info*	*Undesired_events*
+EQ+	p1:real;I	!!source!!	%%constant dest'n%%
+NEQ+	p2:real;I	!!source!!	
+GT+	p3:boolean;O	!+destination+!	
+GEQ+	p4:real;I	!!user threshold!!	
+LT+			
+LEQ+			
+ADD+	p1:real;I	!!source!!	%%constant dest'n%%
+MUL+	p2:real;I	!!source!!	%range exceeded%
+SUB+	p3:real;O	!+destination+!	
(...)			
		Program Effects	
+ADD+	p3 = p1 + p2		
+EQ+	p3 = (p1 =* p2)		
+GEQ+	p3 = (p1 =* p2) OR (p1 - p2 is positive)		
(...)			

* [This equality is precisely defined in a text footnote]

Figure 2-11. An Access Program Table
for Specifying Operations on Real Entities [Clem84].

Notation:

+*name*+ indicates that *name* is an access function.

!!*name*!! indicates that *name* is a term defined elsewhere in the specification.

!+*name*+! indicates (here) that *name* is a value produced by an access function.

%*name*% indicates that *name* is an undesired event that may not be detected until run-time. Double percent signs indicate an undesired event that will be detected at system generation-time.

Except for access functions, each *name* is defined in a dictionary contained elsewhere in the specification.

One product of the SCR Project has been an abstract interface specification method [Parn77b] that evolves traces (in recognition of their frequently-unworkable complexity) by relaxing the restriction against equivalent functions. The SCR specification [Heni78, Brit81a] consists of two partially redundant interface descriptions:

— An *assumption list,* an English-language statement of assumptions implicit in the functional specifications. It contains both basic assumptions underlying module construction and assumptions about "undesired events" (what is to occur in each foreseen incorrect or error situation). The assumption list makes it easier to identify invalid assumptions and is reviewed by both programmers and non-programmers.

— Formal *functional specifications* of programming constructs embodying the assumptions. These can be used directly in user programs. They, in turn, consist of access functions and *events,* i.e., signals from a module to other modules indicating the occurrence of some state change within the first module. Events are "reported via access programs that do not return until the specified conditions hold" [Clem84]. Access functions syntax and semantics are presented in an *access program table* (Figure 2-11).

The assumption list, access program table, and the dictionaries defining the terms and data types used in each are therefore the primary components of an SCR specification.

The SCR specification method has also been employed in other projects (see, e.g., [Hest81]) and has been the subject of several years of both theoretical and practical research. Its chief benefits have been found to be the facilitation of a structuring set of system documentation and the ready implementation of information hiding [Brit83].

2.5 Summary.

Parnas's techniques for abstract interface specification have considerable potential for extension to the VLSI design domain. Like the other techniques discussed in section 2.3.5, they possess the desirable characteristics of being formal, semantic, executable, and black-box. But their chief merit lies in their unusual capability for implementing a precise separation of concerns (information hiding), not only among modules but also vertically along the hierarchy of specification refinement depicted in Figure 2-4. This precision, coupled with Parnas's concern for spareness, suggests that these techniques can serve as a starting point in the development of an abstract interface specification method for VLSI designs that meets the five criteria of section 2.2.

CHAPTER 3

AN ABSTRACT INTERFACE SPECIFICATION APPROACH FOR VLSI DESIGNS

In the preceding chapter, I have examined related contemporary work in the specification of abstract interfaces for VLSI designs and have pointed out the shortcomings of existing VLSI design interface specification methods. This chapter reports on a new approach to such specification that addresses some of these shortcomings, presenting the approach by describing a method that uses it. Chapter 4 contains an evaluation of the method's utility and effectiveness, and chapter 5 provides conclusions and suggestions for future research.

Chapter 3 is organized as follows. I begin by describing an IC module abstract interface specification method, an extension of Parnas's interface specification techniques to the domain of VLSI design. The second section provides a detailed example of how the new method can be used to specify the interface of an IC module. Next, I show how the behavior of compositions of such specifications can be predicted using Communicating Sequential Processes [Hoar78]. A final section focuses on simplifications to the method that can be obtained by exploiting certain characteristics of the VLSI design process.

3.1 Overview and Description of the Approach.

As has been seen, a critical purpose of a VLSI design interface specification is to serve as a *frame* for subsequent design verification. In attempting to develop a specification method that fulfills this purpose, however, one encounters contradictory constraints. On the one hand, it is desirable to construct the specification early in the design process, to capture high-level design concerns and to guide subsequent design. However, in practice even the most top-down design turns up subtle contradictions among the constraints, requiring backtracking. What is needed, and what I propose, is an interface specification approach that can *start* by capturing high-level concerns but then can be *refined* into a more particular specification as further information becomes available over the course of the design. Such a specification fulfills the objectives (section 2.2) of adequacy and malleability; it can serve as a verification frame throughout the design life-cycle, because it can "grow" with the design.

Another objective addressed in this research has been the perceived cost-effectiveness of interface specification at VLSI complexity levels. As has been stated, clear-box specification methods do not scale well to VLSI levels because their complexity must increase directly with the number of components in the circuit. Several black-box techniques have been proposed, but few include abstraction mechanisms that effectively reduce complexity by crisp partitioning of concerns. In the approach I propose, precise partitioning of design function is achievable, and complexity can be further reduced by partitioning timing from function. Additionally, the proposed partitioning mechanisms can be used for geometric as well as for functional/electrical information, enhancing the approach's spareness and cost-effectiveness.

Finally, the proposed approach includes techniques that support the objective of change management by precisely quantifying the concept of *specification adherence*. Such techniques permit design maintainers to measure the degree to which a candidate replacement module meets its specification and thus to predict how much change, if any, will be introduced beyond the module boundary by its inclusion in the design. If such techniques exist, they have not before been applied to change management in VLSI design.

The following paragraphs provide an overview of the proposed approach. In general, this approach extends Parnas's abstract interface specification techniques for software, providing specification refinement and complexity management in a hierarchy of virtual machines. A specification language is not part of the proposal, and the notation and syntactic constructs provided are not purported to be superior. Instead, effort has been focused on identifying and studying the issues involved in specifying abstract interfaces for VLSI designs. The proposed approach, therefore, is an attempt to address these issues generically, and the techniques proposed herein represent a class of abstract interface specification methods.

3.1.1 Model.

The proposed abstract interface specification model is a simple "black box," consisting of an abstract model of a *state* (Figure 3-1). The state consists of a finite set of entities called *state variables* and is accessed through a distinguished subset of those variables called *pins;* state variables that are not pins are called *internal.* Data enter and leave the module, via the pins, in *signal variables.* This model corresponds, of course, to the state machine model used by Parnas [Parn72a], Bartussek [Bart77], Barrow [Barr84], and others [Lisk75] in the treatment of abstract data types in software, where by "abstract data type" is meant a set of objects together with a

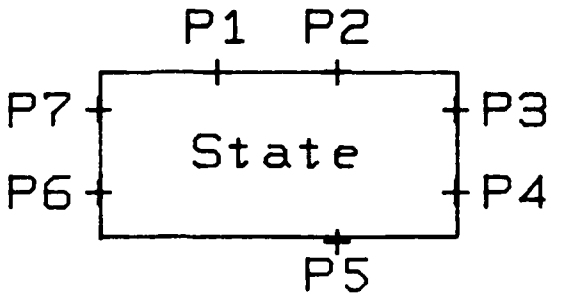


Figure 3-1. Basic Specification Model.
(Pi = Pin i, i = 1,...,7)

set of operations on those objects.

3.1.2 Data Typing.

Indeed, each pin and internal state variable in the model depicted in Figure 3-1 is a particular instance of a single generic abstract data type, called here the *generic state type*. The objects in the generic state type are signals; the operations on the signals vary, but typically include transmission, inversion, selection, and so on. The generic state type is generic because its particularizations depend on the specification of values for semantic and syntactic *attributes*, which are parameters used in the definition of type operations (semantic attributes) and in the definition of legal type usage from outside the module (syntactic attributes). Pins and internal state differ, therefore, in that internal state variables possess no syntactic attributes; this is their only difference.

Since the pins and internal module state are the only objects in this model, it is in the definition of the generic state type that the *level* of the specification is determined. A specification whose generic state type has simpler syntax or semantics (such as, for example, "signal-type = Boolean") is a high-level specification, whereas a specification whose generic state type has more complex syntax or semantics (such as "signal-type = vector," where the vector

consists of a Boolean functional value plus validity and signal strength values) is a lower-level specification. The dilemma of contradictory constraints (alluded to earlier) here confronts those who would construct specification methods. What kinds of syntax and semantics *should* be included in the generic state type? With too much detail, the specification cannot be constructed until after it is needed in the design process; with too little, the specification becomes rapidly overtaken by emerging information gathered from lower-level design.

3.1.2.1 Data Type Evolution.

The answer, I believe, is not to limit an interface specification to any one level of data typing, but instead to evolve the specification generic state type to increasing levels of detail over the course of the design. In this interpretation, a pin or internal state variable is assigned a high-level type early in the specification process and then, as the design process continues, the generic state type is made more complex so that specification syntax and semantics are strengthened in a natural and necessary way. I call this evolution of specifications by data type particularization *specification refinement* (see section 2.2.2, especially Figure 2-4).

Let us illustrate specification refinement by continuation of the earlier example. Early in the specification process a signal might be assigned the type "Boolean," a type that is clearly a high-level abstraction of the functional and electrical characteristics of its signal. Later, without sacrificing the earlier characterization, the Boolean type for this signal might be extended into a type that consists of four bits of information. One of these is the earlier Boolean type, but a second bit might describe the first bit's validity (i.e., "11" might be a valid "one," whereas "10" might represent an indeterminate state) and the remaining two bits might describe the strength of the signal described by the first two.

It is important (indeed essential) to this development that the particularization of these data types represent an *evolution* (and not a revolution). Frequently, when high-level specifications are found to be lacking in essential semantic information, they are discarded and a new specification is developed. However, this poses the problem of reconciling the new specification with the old one, a manual (hence error-prone) process at best. It is much better to evolve the higher-level specification into the lower-level one and thus to avoid the reconciliation problem. As has been stated earlier, such an evolutionary specification is integrated into the design life-cycle. This benefit becomes even more apparent when it is noted that an evolutionary approach makes it possible to compose modules at a high level of specification with modules at a lower level and still

have the composition be meaningful. Such mixed-level specification can aid significantly in an incremental design or redesign effort.

The concept of evolution is made more precise in the following

Definition.[1] Let <A, OA> and <B, OB> be data types, in which the first component of the ordered pair is a set of objects and the second is a set of operations on those objects. Then <B, OB> is an *evolutionary particularization* (or *evolutionary refinement*) of <A, OA> iff there are two mappings ϕ: A→B and ψ: OA→OB such that for all $o \in OA$ and for all $a \in A$

$$\psi(o)(\phi(a)) = \phi(o(a)).$$

This definition is illustrated in Figure 3-2.

A familiar example of an evolutionary refinement is the commonly-understood refinement of the data type "integer" into the data type "real." Real operations on integers do not give results inconsistent with those of counterpart integer operations on integers.

3.1.2.2 Data Type Dimensionality.

Whereas software interfaces are usually characterized in the single dimension of function, IC interfaces traditionally are recognized to contain information in three dimensions: the functional, geometrical, and electrical. These axes do not seem to be orthogonal; instead, an IC module's electrical information can be considered a refinement of its functional information, and so the functional and electrical axes can be collapsed together. Therefore, pins and state variables could

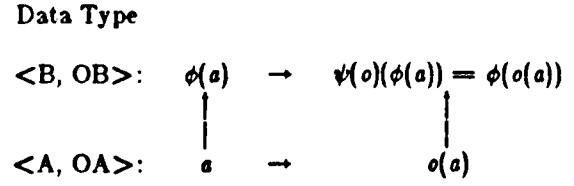


Figure 3-2. Evolutionary Data Type Refinement.

[1] This definition extends the refinement concept developed by the IBM Federal Systems Division [IBM82, Witt85]

be characterized by data types with functional/electrical and geometric components.

However, this traditional characterization leads to unnecessary complexity; an alternate characterization reduces complexity by allowing more precise partitioning of concerns. Recall that dynamic verification is required only for module semantics and that static checking suffices for module syntax. It happens that much of the geometric information in a design can be classified as syntactic, and much of the functional/electrical information is semantic. (This correspondence is not complete: designs possess semantic geometric information (e.g., current density) and syntactic functional/electrical information (e.g., whether a signal is used for input or output)). But, recognizing that the purpose of a specification is to aid in verification, it is more efficient to partition specification elements along syntactic and semantic axes than along functional/electrical and geometric axes. Consequently, in this method generic state types exist at discrete levels of semantic and syntactic complexity, and the particular specification state type, hence the level of the specification, is fixed by a pair

<semantic_level, syntactic_level>.

Here are some well-known examples of semantic and syntactic levels:

a. Semantic. At lesser levels of specification *refinement*, *generic* state types have semantics on the functional end of the functional/electrical spectrum. Functional/electrical values here are typically drawn from a finite set, of size two (Boolean signal levels), three (Boolean with the indeterminate signal level often characterized as "X," or more (e.g., twelve, obtained by multiplying three with a set of four signal strength values, such as floating strength, gate-driven strength, steered strength, and unknown strength). Note that the three-value type is an evolutionary refinement of the two-value type, and the twelve an evolutionary refinement of both.

Further on, one typically requires generic state types with performance semantics, containing signal types consisting of vectors of values that specify signals in terms first of voltage levels and then of capacitive and resistive drive, current levels, power density and consumption, and so on.

b. Syntactic. Here, in evolutionary order, are three sample syntactic types that might be used in pin specification:

i. An "unspecified" syntactic type that imposes no constraint whatsoever on pin placement along the module periphery.

ii. An "order-specified" syntactic type that constrains a pin to a given position in an ordering of the pin set around the periphery of the module.

iii. A "location-specified" syntactic type that constrains a pin to a given geometric window on the boundary of the module.

3.1.3 Access Functions and Specification Refinement.

Once again extending the techniques of Parnas (section 2.4), I characterize each module pin by associating with it an *access function*. Such access functions can be of one of two types: *value* or *effect* [Clem84]. Value access functions are used to obtain a value on a pin; effect access functions are used to set internal state of the module using the value on a pin. I contend that VLSI modules can be fully specified using such access functions, and that such a specification permits efficient change management over the course of the VLSI design process.

The following definitions formalize those of section 2.4 and extend them into the IC design domain:

Definitions. Let an IC module have a set S of internal states, and let R be a set of input or output signal values. Then a *value access function* is a function

$$v: S \rightarrow R,$$

and an *effect access function* is a function

$$w: S \times R \rightarrow S.$$

From these definitions, the notion of (semantic) specification refinement can now be formalized. (A similar formalism can be drawn for syntactic refinement.) Let V and W be the sets of value and effect access functions associated with the pins of a given module. Let us first identify, for each (value access) function v_i in V, a corresponding *signal range* r_i, such that

$$v_i : S \rightarrow r_i, \; i = 1, 2, ..., \text{card}(V).$$

Similarly, for each (effect access) function w_i in W, let us identify a corresponding *signal domain* d_i, such that

$$w_i : S \times d_i \rightarrow S, \; i = 1, 2, ..., \text{card}(W).$$

These signal range and domain sets encompass the spectra of values that pin signals can attain.

Observe, therefore, that the locus of specification refinement is in the signal range and domain sets r_i and d_i and in the state set S. By our previous discussion of state variable semantics, for a given i each element of r_i, d_i, or S may be seen to be a vector in SE, where SE is a set of semantic values. Then *(semantic) specification refinement* is the redefinition (by enlargement) of the sets r_i, d_i, and/or S, for any of the access functions, and/or for the state set, which make up the specification.

An example may help. Suppose, for pin 1 in a given module, $SE_1 = \{0,1\}$. This description corresponds to pure behavioral simulation using two-valued logic. If one wished to refine the specification to three-valued logic, he would represent this refinement as a redefinition of SE_1 to $\{0,1,X\}$.

As another example, a module's internal state might consist of vectors in

{strong 0, strong 1, weak 0, weak 1}.

A specification refinement might redefine this set to

{floating, unknown, steered 0, steered 1, gate-driven 0, gate-driven 1}.

3.1.1 Scheduling.

How should scheduling be handled by this specification method? This important question can be addressed in at least two ways. A first way is to add a separate temporal dimension to the generic state type semantics, so that pins and internal state variables could have scheduling semantics in addition to functional/electrical (and possibly geometric) semantics. This could lead, for example, to a state type data structure consisting of a Boolean signal level, a signal strength, and a clock phase during which the signal is active. Bear in mind, however, that we have been seeking to control specification complexity by precisely separating concerns. Adding a temporal dimension to data type semantics detracts from this complexity control objective by mixing two potentially separable kinds of information, i.e. functional/electrical and scheduling semantics. Indeed, from the viewpoint of local (module) functional/electrical concerns, clocked signals are not semantically different from unclocked signals: scheduling information, if any, is external (global). Including global information in a module interface specification violates information hiding by including knowledge of the module's context.

A different, albeit less conventional, way to handle scheduling is to treat scheduled signals semantically the same as unscheduled signals and to add scheduling information instead to the *syntactic* portion of the specification. Because clock signals, for example, change the state of a module just as other signals do, one can define access functions corresponding to clock inputs and outputs just as for other pins. Scheduling itself then becomes a global concern and, as such, is dealt with syntactically at the system integration level. This method of treating scheduling achieves separation of concerns and is a substantial conceptual simplification. As will be shown in section 3.3, the synchronization and scheduling of individual access functions within a module can be partitioned from the description of access function semantics, which can in turn then be cast as independent (asynchronous) sequential processes.

3.1.5 Specification Adherence and Tolerance.

What does it mean to say that a module *adheres* to such a specification? The association of module semantics with access functions enables us to relate the concept of adherence to access function terminology.

I will formalize the concept of adherence by appealing to the fuzzy set theory pioneered by L.A. Zadeh [e.g., Zade75, Zade84]. Fuzzy sets are sets for which set membership is not defined solely by an in-or-out criterion. Instead, Zadeh defines a fuzzy set *membership function*

$$\mu[F] : A \rightarrow [0,1],$$

that indicates, for each element of a set A, its *grade* or *degree of membership* in a fuzzy set F. Often, when A is a continuous interval, $\mu[F]$ is a monotonic function over A (Figure 3-3).

In software design, in which functional adherence is the only real concern (section 2.2.2), adherence can be defined with traditional set theory. That is, for a given module specification, there will be a set of modules that adhere to the specification, and a complementary set of modules that do not. But in hardware design, adherence is not so sharply characterized, for the reason noted in section 2.1: "Circuits must not only work, they must *perform*." There must be a continuum of the utility of specification adherence that corresponds to the continuum of performance. Consequently, the set of modules that adhere to a specification is a fuzzy set. I will further introduce for each specification *adherence functions*, analogous to membership functions, which map modules onto the interval [0,1] according to the utility of the degree to which they satisfy the specification.

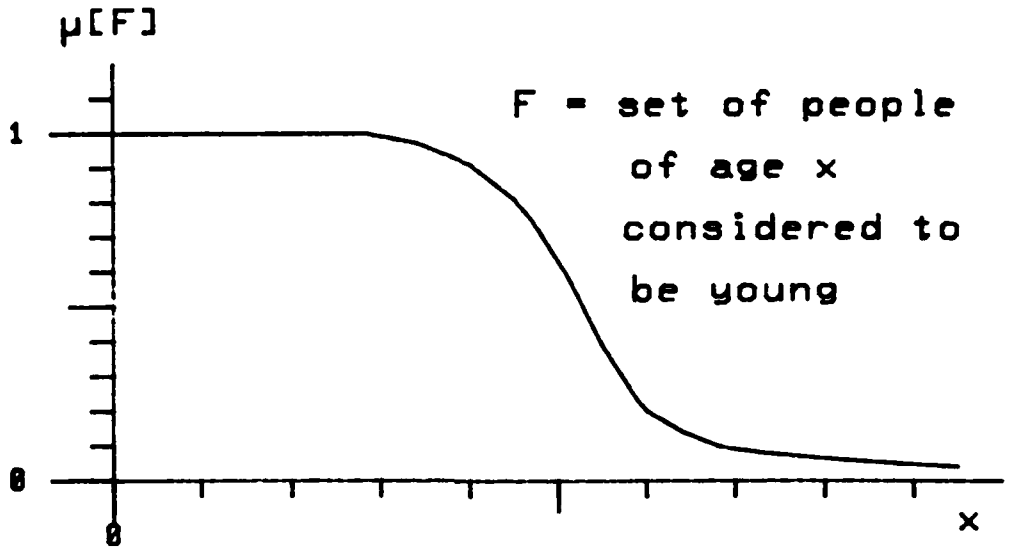


Figure 3-3. A Fuzzy Set F
and its Membership Function $\mu[F]$ *[Zade84]*.

Unlike fuzzy sets, however, which possess a single membership function, an IC interface specification must have multiple adherence functions, one for each of the multiple points at which modules may or may not adhere to the specification. Happily, one adherence function suffices for each access function (and hence for each pin). Unhappily, this adherence function is in general much more complex than the membership function of Figure 3-3. In fact, the adherence function can be up to four-dimensional, in that the adherence function maps each (possibly two-dimensional) input and output of its corresponding access function into the range [0,1] (which range is here always represented on the y-axis (ordinate) by convention).

For analysis, however, the chief concern is with the two-dimensional projection (of the adherence function) that corresponds to each separate access function input value, so that module output is graphed on the x-axis (abscissa) and the adherence function value on the y-axis. On the y-axis of such a projection (e.g., Figure 3-4), the designer can choose *intervals of adherence* for the specification, with each interval assigned an intuitive meaning or figure of merit. The intervals

for the adherence function output values that are intercepted on the x-axis by these adherence intervals are commonly referred to as the *tolerances* for these outputs. This tolerance concept, if not often formalized, is also familiar. For example, the "plus and minus x%" color banding of resistors is well-known in electrical engineering. Even in software, where the concept is almost always abstracted away, one must sometimes acknowledge tolerances, as when testing a floating-point variable for zero, e.g.

if (abs(x) < tol_value) *then* ....

The ordinate of the adherence function graph is not dimensionless, as has previously been suggested. Actually, the adherence function measures the *marginal utility* of reduced tolerance. In so doing, it provides a means, heretofore unavailable, for the specifier/designer to communicate to the implementer the cost of suboptimization. As such a vehicle, the adherence function might be termed an *adherence value* function. Further research is needed to exploit this important concept to optimize its value in the VLSI design process.

Adherence functions can be formalized as follows. Let

$$v_i : S \rightarrow r_i$$

be a value access function in a specification. The specification will then also include an adherence function

$$\alpha[v_i]: S \times r_i \rightarrow [0,1]$$

such that, *if* $v_i(q0) = x0$, $\alpha[v_i](q0,x)$ equals 1 *if* $x = x0$ (the specified value) and some other value in [0,1] if $x \neq x0$. The variable x represents an output signal, and thus it may be multi-valued (a vector).

Similarly, if

$$w_i : S \times d_i \rightarrow S$$

is an effect access function in a specification, the specification also includes an adherence function

$$\beta[v_i]: (S \times d_i) \times S \rightarrow [0,1]$$

such that, if $w_i(q0,x0) = q1$, $\beta[w_i](q0,x0,q)$ equals 1 if $q = q1$ and some other value in [0,1] if $q \neq q1$. Here the variable x0 represents an input signal; it too can be a vector.

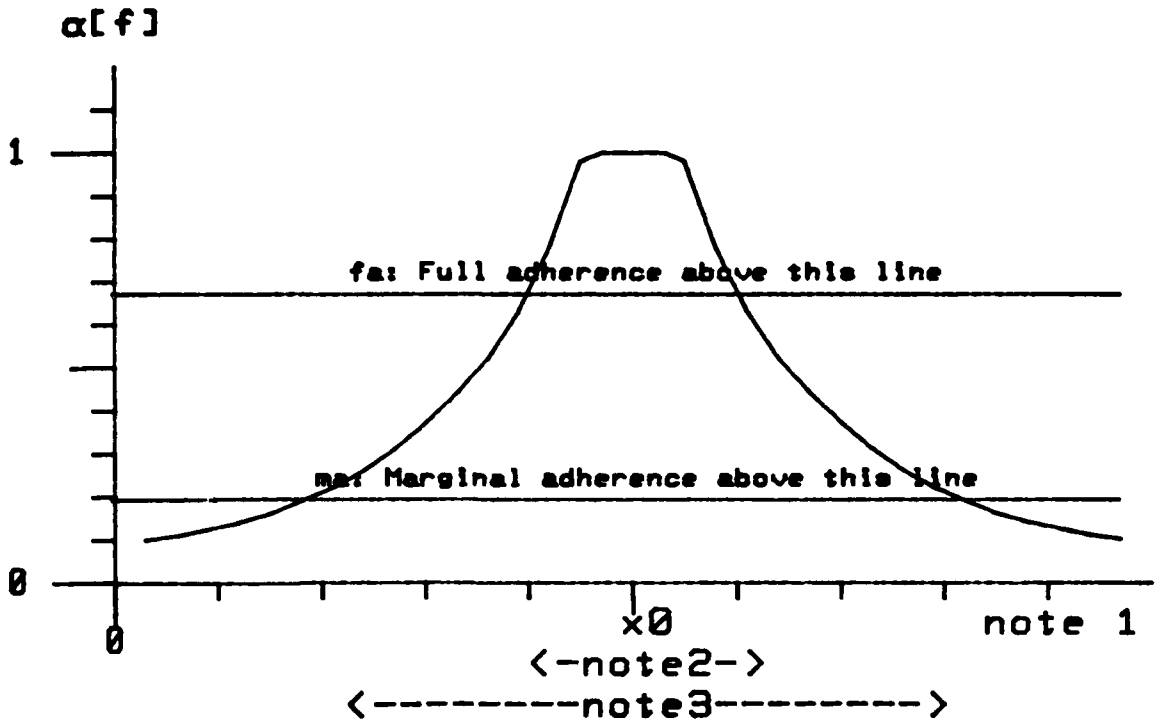


Figure 3-4. A Projection of an Adherence Function $\alpha[f]$ for a Single Input of an Access Function f.

Notes:

1. X-axis (abscissa) measures output, state output (if f is an effect access function), or signal variable output (if f is a value access function). x0 is the output value predicted by the specification access function f.
2. Variable tolerance range for full adherence.
3. Variable tolerance range for marginal adherence.
4. The values fa and ma are chosen by the designer; they are provided in this figure only to illustrate sample meanings of the intuitive concepts of "full" and "marginal" adherence.

Adherence functions appear to be complicated to use. Part of their complexity may result from the recognized detail explosion that has hampered efforts toward developing techniques for abstract interface specification of VLSI designs at refinement levels below the functional. But adherence functions exist even at the functional level. In functional specifications, adherence functions are degenerate, however, and return zero rather than some other value in [0,1] for an input other than the specified one (*vide* function 1, Table 3-3). Thus, for functional specifications the access function itself suffices for the adherence function, masking the adherence function's

existence. Identifying the adherence functions is necessary only in more refined specifications; however, if the existence of these functions is acknowledged at the functional level, the specification refinement process can be visualized more consistently. (In section 3.4 some ways will be suggested to simplify adherence function usage without sacrificing precision.)

The definitions of "abstract interface specification" and "adherence" can now be completed, as follows:

Definition. An IC module *abstract interface specification* is a 6-tuple <P, S, V, W, VAF, WAF>, where

P is a set of *pins*,

S is a set of *internal states*,

V is a set of *value access functions*,

W is a set of *effect access functions*,

VAF is a set of *value adherence functions*, and

WAF is a set of *effect adherence functions*.

In passing, we should recognize also the value of informally including in the specification a set of English-language *assumptions* [Heni78, Brit81a]. These assumptions (see section 2.4.3) provide no essential abstract interface information, but they do set the context for the formal content of the specification and do so in a way that can be easily assimilated by reviewers. In addition, they play an important role in communicating the content of the specification to non-technically-oriented reviewers, who may, thus enlightened, be able to provide valuable feedback.

If a specification's adherence functions are constructed appropriately, a consistent interpretation of the concept of adherence can be applied not only across all inputs of a single access function, but indeed across all access functions in the specification. The result will be that, for each k, the situations in which adherence functions return values in the interval [k,1] have a common meaning.

Definition. Let MS = <P,S,V,W,VAF,WAF> be a module specification, and let M be an IC module. A pin p of M is said to k-*adhere* to specification MS if

(1) when p corresponds to an access function v ϵ V, for all q0 ϵ dom(v), if M is in state q0 and if x is the output when v is applied to M, then

$$\alpha[v](q0,x) \geq k;$$

and

(2) when p corresponds to an access function w ϵ W, for all pairs (q0,x0) ϵ dom(w), if M is in state q0, if x0 is the input to M, and if this causes M to make a transition to state q, then

$$\beta[w](q0,x0,q) \geq k.$$

Further, module M K-*adheres* to specification MS if K is the smallest k for which any pin p in M k-adheres to MS.

Some important corollaries of these definitions can be noted.

(1) If k-adherence is to be established by simulation, exhaustive simulation is required. Modules must therefore be kept small to avoid the penalties of the combinatorial explosion. Fortunately, having small modules also contributes to intellectual control, and systems of hundreds of small software modules have been found to be easier to specify and manage than have systems of tens of larger modules [*Brit81b*, *Parn83b*, *Parn84*].

(2) However, k-adherence need not be established by simulation. If a module's behavior and performance can be predicted analytically (e.g., see [Barr84]), the definition can be tested using these means.

(3) Adherence to a functional specification is 1-adherence. This concept is meaningful because 1-adherence and its implicit adherence functions are well understood. To provide the same degree of intuition for k-adherence, the specifier must construct adherence functions carefully to reflect (a) the degree of tolerance in a specification and (b) the impact of incomplete adherence. For example, if a "cushion" is intentionally built into a specification, the adherence function should permit a wide degree of deviation before it falls from a value of 1. As another example, pin behavior may fall off as the square of a voltage deviation but only linearly with current deviation. The adherence function should reflect this. Research is needed into the development of adherence functions that both meet these criteria and also yield values that lend meaning to k-adherence across a wide spectrum of designs.

3.2 Example.

An example will clarify the distinctions of the proposed interface specification technique. Figure 3-5 provides, using a clear-box specification method, a specification of a register cell used in the recent development of the WAfer-Scale Systolic Processor [Hedl83, Hedl84b, Cole85] at the University of North Carolina. The logical operations specified for this cell are SHIFT, invoked by a value of 1 on the *sh* signal, and HOLD, invoked by a value of 0.

Figure 3-6 shows how this same module is specified in the proposed method. The major sections of the specification are as follows:

(1) *Assumptions.* This informal list states in prose the fundamental assumptions underlying the module's definition. As in Parnas's method, the assumption list can contain statements about required subsets, expected changes, and behavior in the face of undesired events.

(2) *Specification Level.* Identification of the syntactic and semantic levels of the specification's generic state type.

(3) *Pin and Internal State Variable Summaries.* This list contains: (a) the names of the module pins and internal state variables; (b) the identity of the access function corresponding to each pin; and (c) a "call/definition" indicator that tells whether the access function is defined in this specification or merely called.

(4) *State Variable Attribute Initializations.* The generic state type requires particularization by the initialization of its *attribute* values. For example, the generic state type might define state variables with capacitive load C; in this section, instances of this type would be particularized by the assignment of a value to C.

(5) *Access Function Definitions.* An abstract statement of the actions to be performed on invocation of each access function. To enhance robustness of the specification under refinement, this statement should be as data-type invariant as possible.

NAME WASP 1.0, July 15, 1984

plashin — LSSD input driver

Variations: plashina

SYNOPSIS

LSSD memory element: static memory cell that can convert to a shift register.

USAGE

This is the basic shift register memory cell used in both the MinAlu and the Switch.

PROPERTIES

Size delta x = 18 lambda delta y = 90 lambda

Power Consumption (to be determined)

PINS

p2sh	input	active phi2	restored	control
p2shbar	input	active phi2	restored	control
p1shbar	input	unclocked	restored	control
scin	input	active phi2	steered	scan data input
scout	output	active phi2	steered	scan data output
routbar	output	unclocked	restored	data output

DESCRIPTION

Static memory cell that can function as a shift register. Typically controlled by a single shift signal, sh,

sh=0 static memory

sh=1 shift register

This usage assumes the following logic for the control signals:

p2sh = phi2 AND sh

p2shbar = phi2 AND (NOT sh)

p1shbar = phi1 OR (NOT sh)

Data are shifted from left to right.

NOTES

1) Control signals can be generated by cell rwcntl.

RELATED DOCUMENTATION

See manual pages for rwcntl and LSSD memory.

TESSELLATION

Cells abut horizontally to form registers of arbitrary length that shift from left to right.

ORIGIN

Originally derived from Stanford Cell Library (old version) cell PlaShiftIn (CIF ID 81). Minor modifications by Hedlund (UNC) and Lospinuso (UNC).

LOGIC DIAGRAM

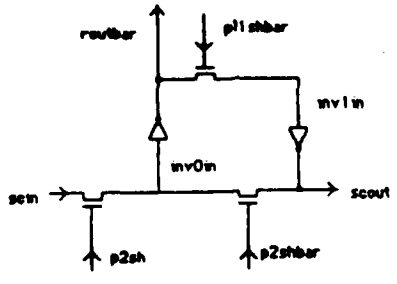


Figure 3-5. A Clear-Box Cell Specification [Hedl84a].

(6) *Data Type Definitions.* Here, the generic state type's objects and operations are defined. These definitions are level-specific, but they can be reused.

(7) *Adherence Function Definitions.* In this final section, an adherence function is formally defined for each access function in the specification. These definitions are also level-specific.

plashin
Module Specification

ASSUMPTIONS

Synopsis: LSSD memory element: static memory cell that can convert to a shift register.
Size: delta x = 18 lambda; delta y = 90 lambda
Description-01: Data are shifted from left to right.
Description-02: p2sh = p2 AND sh; p2shbar = p2 AND (NOT sh); p1shbar = p1 OR (NOT sh)
Notes-01: Control signals can be generated by cell rwcntl.
Related_Documentation: See manual pages for rwcntl and LSSD memory.
Tessellation: Cells abut horizontally to form registers of arbitrary length that shift from left to right.
Origin-01: Originally derived from Stanford Cell Library cell PlaShiftIn (CIF ID 81).
Origin-02: Minor modifications by Hedlund (UNC) and Lospinuso (UNC).

SPECIFICATION LEVEL[2]

Semantic: 4-strength
Syntactic: immaterial

PINS

Name	AccFn	C/D
scin	G_ScanIn	C
scout	P_ScanOut	D
routbar	P_DataOut	D
p2sh	S_Shift	D
p2shbar	S_Hold	D
p1shbar	S_Recirculate	D

INTERNAL STATE VARIABLES

left
right

STATE VARIABLE ATTRIBUTE INITIALIZATION

None.

ACCESS FUNCTIONS

G_ScanIn: none

[2] See Table 3-2.

Figure 3-6. Proposed Specification for *plashin*.

```
P_DataOut:
#/*************************************************************/
#/*     Access function P_DataOut:
# */
! routbar = ROUTBAR → skip;  /* Null access function */

P_ScanOut:
#/*************************************************************/
#/*     Access function P_ScanOut:
# */
| ! scout = SCOUT → skip;

S_Shift:
#/*************************************************************/
#/*     Access function S_Shift:
# */
| ? P2SH = p2sh →
# SS = signal(P2SH);
# ST = signal_threshold(P2SH);
  | SS >= ST →
        ? SCIN = scin;
#       steer(SCIN,P2SH,left);
#       inv(left,ROUTBAR);
  | SS < ST → skip;
  |

S_Hold:
#/*************************************************************/
#/*     Access function S_Hold:
# */
| ? P2SHBAR = p2shbar →
# SH = signal(P2SHBAR);
# ST = signal_threshold(P2SHBAR);
  | SH >= ST →
#       steer (SCOUT,P2SHBAR,left);
  | SH < ST → skip;
  |

S_Recirculate:
#/*************************************************************/
#/*     Access function S_Recirculate:
# */
| ? P1SHBAR = p1shbar →
# SR = signal(P1SHBAR);
# ST = signal_threshold(P1SHBAR);
  | SR >= ST →
#       steer (ROUTBAR,P1SHBAR,right);
#       inv (right, SCOUT);
  | SR < ST → skip;
  |
```

Figure 3-6. Proposed Specification for *plashin*. (Cont'd)

```
DATA TYPE DEFINITIONS
/* Data Type Definition:
 * Four-strength semantic refinement level.
 * Immaterial syntactic refinement level.
 *
 * At this level the state variable data structure is
 *      typedef struct stvar {
 *              int sigvartyp;
 *              int sigfval;
 *              int sigstrength;
 *              } STVAR ;
 * with
 *      sigvartyp = variable type (4 = four-strength/immaterial);
 *      sigfval = functional value in {0,1,9 (= undefined)};
 *      sigstrength = signal strength in
 *           {0 (= unknown),
 *            1 (= floating),
 *            2 (= steered),
 *            3 (= restored)}
 * with signal strength codes semantically ordered
 *      (i.e., i > j → strength i > strength j).
 */

#define INFINITY 65535

inv(input,output)
   STVAR *input, *output;
{
output→sigvartyp = input→sigvartyp;
output→sigfval =
        (input→sigfval == 9) ? 9 : (1 - input→sigfval);
if (input→sigvartyp < 4) output→sigstrength = 0;
else output→sigstrength = (input→sigstrength >= 2) ? 3 : 0);
return;
}

int
signal(input)
   STVAR *input;
{
return(INFINITY);
}

int
signal_threshold(input)
   STVAR *input;
{
return(0);
}
```

Figure 3-6. Proposed Specification for *plashin*. (Cont'd)

```
steer(input,control,output)
   STVAR *input, *control, *output;
{
output→sigvartyp = input→sigvartyp;
output→sigfval = (control→sigfval == 1) ? input→sigfval
                : output→sigfval;
if (input→sigvartyp < 4) output→sigstrength = 0;
else output→sigstrength = (input→sigstrength >= 2) ? 2 : 0);
return;
}
```

ADHERENCE FUNCTION DEFINITIONS

Access Fn f	Adherence Function $\alpha[f]$	
P_DataOut	$\alpha(q,x)$	= 1 if x→sigfval = P_DataOut(q)→sigfval and x→sigstrength $\geq$ 2; = 0 otherwise.
P_ScanOut	$\alpha(q,x)$	= 1 if x→sigfval = P_ScanOut(q)→sigfval and x→sigstrength $\geq$ 2; = 0 otherwise.
S_Shift	$\alpha(q,s,q')$	= 1 if q' = S_Shift(q,s); = 0 otherwise.
S_Hold	$\alpha(q,h,q')$	= 1 if q' = S_Hold(q,h); = 0 otherwise.
S_Recirculate	$\alpha(q,r,q')$	= 1 if q' = S_Recirculate(q,r); = 0 otherwise.

Figure 3-6. Proposed Specification for *plashin*. (Cont'd)

Several distinguishing features can be noted in Figure 3-6. In the first place, module semantics are defined on a per-pin basis, rather than globally. Such a partitioning of module semantics makes the specification easier to construct and verify intellectually; it also reduces the combinatorial complexity of the formal verification task in two ways. First, each pin's semantics are only a subset of module semantics, making the verification task simpler at each pin. Also, for IC modules, physical fabrication constraints (i.e., pin count restrictions) will continue to provide motivation to encapsulate module subfunctions, keeping low the number of separate pin verifications [Moor84].

Figure 3-6 illustrates this partitioning. The module functions SHIFT and HOLD have been redefined into the access functions P_ScanOut (which *Puts* a value on the *scout* pin), P_DataOut, and the three qualified clock access functions S_Shift, S_Hold, and S_Recirculate (which set the state variables *left* and *right* according to an input value received on the *p2sh*, *p2shbar*, and *p1shbar* pins respectively). The sixth access function G_ScanIn is not defined in the specification, but appears instead as a function call (? left = ScanIn) in the definition of the function S_Shift. (Access functions have been expressed in the Communicating Sequential Processes (CSP) notation [Hoar78], which will be explained in section 3.3. The question mark in CSP is an input operator.)

Note that as a specification *is refined*, it may be necessary to add additional pins: for example, power and ground pins will generally not appear in less-refined specifications, nor will pins at both ends of busses. Because the specification is defined on a per-pin basis, however, the change required by the addition of new pins is reduced. Power and ground pins generally do not require access functions, as their information of concern is purely syntactic until one arrives at the most refined specification levels; even at these levels, their access functions would be easy to construct. As for the other pins, most specifications contain one reference (definition or call) to an access function for each logical pin in the module. While some pins (e.g., control busses such as p1shbar) may be implemented with more than one physical termination, for less refined specifications the access function is assumed to define the behavior at all physical pin terminations. If it does not, as perhaps for long poly control busses, new access functions can be defined for each termination whose behavior differs. Usually, however, the scope of these access functions in module state is small, and hence a small amount of change, if any, is required in adjacent access functions.

At least one disadvantage accrues from this partitioning of module semantics into per-pin components: global module constraints, such as power consumption, current draw, tessellation, and aspect ratio, are difficult to specify. Possible approaches include (1) apportioning these constraints among the pins and (2) constructing distinguished state variables ("constraint variables"), global to all access functions, that express these constraints. Constraint variables will probably differ in type from the generic state type, complicating the specification, but they will exist in limited numbers. Further research is needed to determine the best way to specify global constraints within the proposed approach.

A second thing to note about Figure 3-6 is that it is executable. Using a CSP interpreter, one can obtain an approximation of module performance at any level of specification definition; such an approximation can be used as a frame for design verification. As an example, Figure 3-7 illustrates the role of a "Specification Interpreter" based on the CSP/84 implementation of CSP [Jaza80a, Jaza80b, Midd84, Midd85] which was used experimentally in this research to approximate the behavior of CSP access functions (see Appendix B).

Third, the specification in Figure 3-6 is malleable; that is, by undergoing specification refinement it can continue to serve as a frame for verification throughout the design life-cycle. Ideally, such refinement affects only the data type and adherence function definitions: there should be no need to change access function definitions, a fact that strengthens the robustness of the specification under refinement. Furthermore, if specification refinement is evolutionary, compositions of modules specified at different levels of refinement can be interpreted consistently. For example, the definitions of the operators 'inv' and 'steer' have been enriched to carry out run-time typing of their inputs, and thus they can process variables of less semantic complexity. The added overhead of such run-time typing is justified for specification interpretation, which need not be extremely efficient; and the added complexity that run-time typing adds to data type definition can be justified because data type definitions, once completed, are reusable and can be catalogued.

Fourth, the specification in Figure 3-6 is both adequate and precise. Adequacy is guaranteed through specification refinement, which in turn can be accomplished by evolutionary refinement of the generic state type. Precision exists at all specification levels through the formal definition of access functions, the generic state type, and adherence functions.

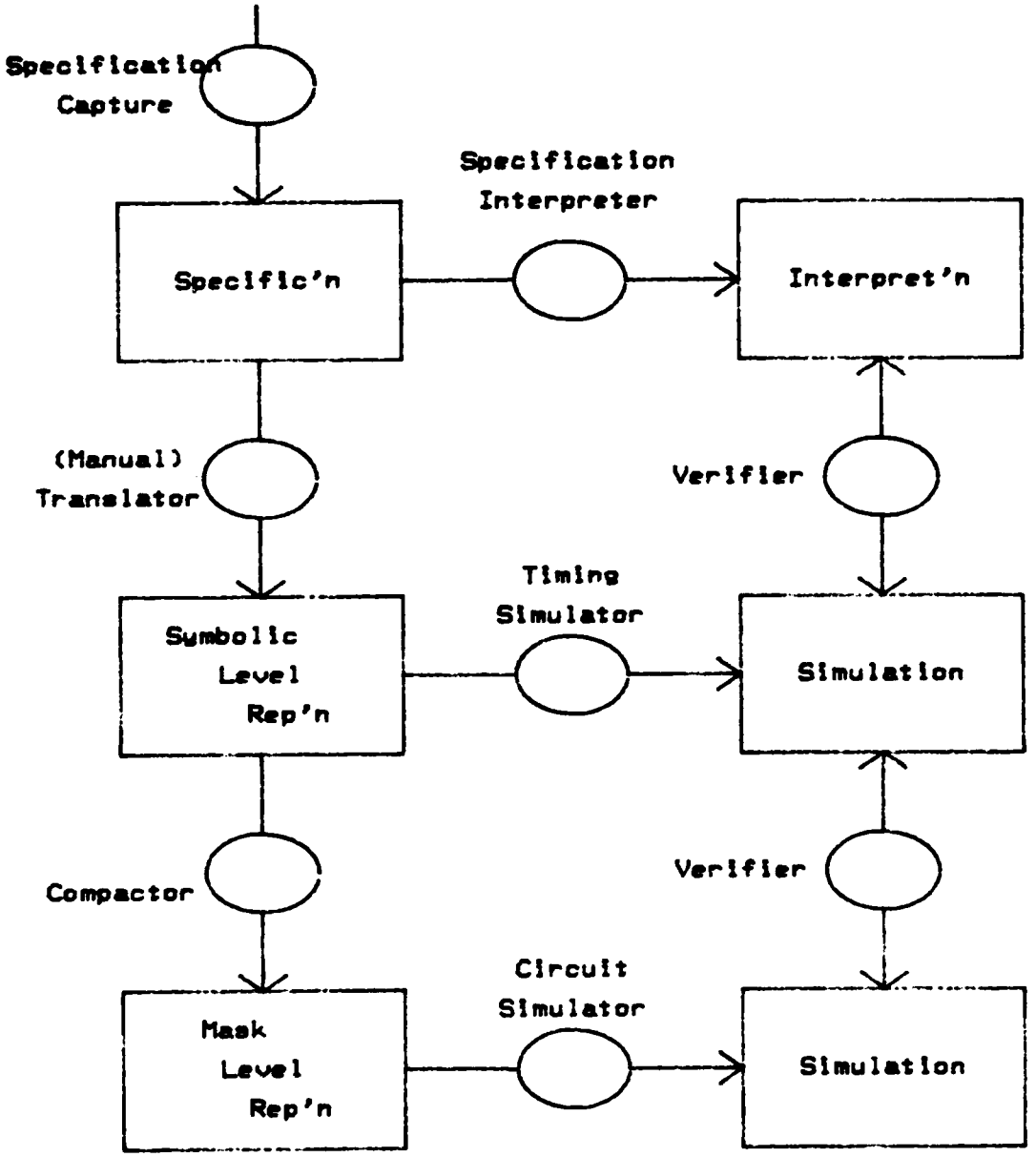


Figure 3-7. Role of Specification Interpreter
in the VLSI Design Process.

(Design representations are listed in the left column, intermediate results for verification purposes in the right.)

Finally, the specification in Figure 3-6 is long — four times as long as the clear-box specification in Figure 3-5. This length is the cost incurred for the benefits of clean partitioning and facilitated verification, executability, malleability, adequacy, and precision. Section 3.4 will contain a discussion of the prospects for reducing this cost. First, however, I shall examine in more detail some issues involved in composing interface specifications that are constructed in the proposed manner.

3.3 Composition of Specifications.

3.3.1 Synchronization and Scheduling.

As has been noted, a chief advantage of a black-box specification method is its clean separation of architectural and implementational concerns. However, the complexity of a large black box cannot be reduced by resorting to descriptions of its internal hierarchy, for to do so would make unintended assumptions about its implementation. Instead, I seek to reduce black-box complexity in two ways: first, by providing an abstract model of its operation; and second, by partitioning its semantics and specifying them on a per-pin basis rather than globally. Because the entirety of the abstract state model might be large, it is usually undesirable that any pin effect access function should be required to update any significant portion of the state. Instead, these access function effects should be localized to reduce function complexity and to improve intellectual control, aiding verification.

This requirement necessitates the introduction of some mechanism for sequencing the application of access functions. A simple example illustrates the problem. Suppose a module has three state variables, v_1 , v_2 , and v_3 , and two effect access functions, f_1 and f_2 , where the effect of any call to f_1 moves the value in v_1 to v_{1+1}. Obviously the call sequence f_1 , f_2 produces a potentially different result in v_3 than does the sequence f_2 , f_1. At the same time, to maintain simplicity one does not wish to involve f_1 with testing either the value in v_3 or the status of invocation of f_2. Another expedient is required.

Put another way, a mechanism is needed to schedule and synchronize the invocation of dependent sequences of independent actions in a way that produces consistent and predictable results. In a landmark paper [Hoar78], Hoare proposed a notation, called Communicating Sequential Processes (CSP), which integrated previous work on this problem. In Hoare's words, the essential proposals of CSP are the following:

> (1) Dijkstra's guarded commands are adopted ... as sequential control structures, and as the sole means of introducing and controlling nondeterminism.
> (2) A parallel command ... specifies concurrent execution of its constituent sequential commands (processes). ... They may not communicate with each other by updating global variables.
> (3) Simple forms of input and output command ... are used for communication between concurrent processes.
> (4) Such communication occurs when one process names another as destination for output *and* the second process names the first as source for input. ... There is *no* automatic buffering: in general, an input or output command is delayed until the other process is ready with the corresponding output or input. Such delay is invisible to the delayed process.
> (5) Input commands may appear in guards. A guarded command with an input guard is selected for execution only if and when the source named in the input command is ready to execute the corresponding output command. If several input guards of a set of alternatives have ready destinations, only one is selected and the others have *no* effect, but the choice between them is arbitrary. ... [Hoar78]

Following another suggestion of Parnas [NPS79], I chose CSP as a notation for expressing scheduling and synchronization for the proposed specification method. The alternative, which is used in Simula, ClassC, and some commercial IC simulators, is to use an event list driven by a global clock. Such usage focuses the specification around a global issue, the passage of simulated time, hindering the localization of concerns that facilitates change management. CSP, on the other hand, explicitly separates synchronization and scheduling from the description of process function. The potential for reducing the complexity of change through such a separation of concerns is just beginning to be noticed in the VLSI research community (see, e.g., [Lieb85]).

To use CSP in IC module specification, one collects module access functions into a single CSP process, which corresponds to the module and is called the "module process." The (CSP) module process incorporates elements of the module specification in the format shown in Table 3-1. As can be seen, the module process contains all the module specification except for the level-specific information: data type definitions (which could be included, but which are generally omitted for brevity) and adherence function definitions.

The core of the module process is the access function definition list. As Table 3-1 indicates, this list consists of a sequence of guarded definitions of the form

<pin_guard> → <statement_list>

As has been stated, in a specification there is a one-to-one relationship between module pins and access functions; however, not every pin has an access function definition in the module process. Instead, definitions are included only for (1) *input* pins of *effect* access functions, and (2) *output* pins of *value* access functions. The remaining pins are referenced from within these definitions as access function *calls*. A "Call/Definition (C/D)" indicator has been included in the pin summary (see Figure 3-6) to reflect whether an access function call or definition is included in the specification.

<module_process> ::=
    process module_name ::
    <pin_declaration_list>
    <state_var_declaration_list>
    <access_function_defn_list>
    end process

<pin_declaration_list> ::=
    <pin_declaration> | <pin_declaration> <pin_declaration_list>

<pin_declaration> ::=
    {*guarded*} [*input* | *output*] *port* <sig_type> pin_id ;

<sig_type> ::= (Varies with specification refinement level. See data type definitions for the particular level desired.)

<state_var_declaration_list> ::= <state_var_declaration>
    | <state_var_declaration> <state_var_declaration_list>

<state_var_declaration> ::= (Varies with specification refinement level. See data type definitions for the particular level desired.)

<access_function_defn_list> ::=
    <state_var_initialization_statement_list>
    *[<guarded_defn_list>]

<VB> ::= |

<guarded_defn_list> ::=
    <guarded_defn> | <guarded_defn> <VB> <guarded_defn_list>
        (Note that a vertical bar, <VB>, must separate guarded definitions.)

<guarded_defn> ::= <pin_guard> → <statement_list>

<pin_guard> ::= ? state_var_id = pin_id
    | ! pin_id = state_var_id

Table 3-1. Partial Syntax of a CSP/84 Module Process [Jaza80a, Jaza80b, Midd85].

(Expansions for <state_var_initialization_statement_list> and <statement_list> can be obtained from the references.)

Segregating the semantics of synchronization and scheduling into the pin guard separates them from the access function's behavior. Thus, the module process body consists of an infinite *polling loop* (represented by the notation *[<guarded_defn_list>]), a loop that tests each of the pin guards non-deterministically to determine whether a call has been made of that access

function from outside the module. If a call has been made, the <statement_list> associated with the access function is executed, including any embedded access function calls; otherwise, polling continues. In Figure 3-6, the outer syntactic brackets of this loop have been omitted, but the vertical bar that separates access function definitions has been retained to show how the access functions making up the <guarded_defn_list> are intended to be concatenated into this list.

Returning to the question posed at the beginning of this section, then, how are access functions to be scheduled in a consistent and predictable way using CSP? No additional mechanism is needed. One simply introduces in composition other CSP module processes that schedule access function executions through appropriate call sequences. In Figure 3-6 such an external call is illustrated: the definition of access function S_Shift contains the statement

? SCIN = scin;

which is a call of the (implicit) value access function G_ScanIn (for "Get ScanIn"). As part of the module composition, the *scin* input pin has been matched (through a separate set of port-to-port connection declarations called a *channel file*) with an output pin, say *P*, on an adjacent module. When G_ScanIn is called in the execution of function S_Shift, a call is immediately issued through the channel file correspondence to the (value) access function for pin P, which in turn executes and returns a value to the *scin* pin. This value is then input to the state variable SCIN to complete the "? SCIN = scin;" function call, and access function S_Shift proceeds.

The module process is therefore an information hiding module in Parnas's sense of the term. It accomplishes the objective of separating architectural and implementational concerns, because access functions are only an abstract representation of module behavior. Moreover, the module process is attractive as a specification because synchronization and scheduling concerns are clearly separated from access function definitions. Finally, because this separation has been made, and because module behavior is specified on a per-pin basis, individual specifications are small and therefore easily comprehended and verified, even with exhaustive simulation.

Before proceeding to discuss the composition of specifications further, I will enumerate some issues I encountered in using CSP as an interface specification notation in general and as an access function scheduling and synchronization mechanism in particular.

3.3.1.1 Power of CSP as a Notation for Interface Specification.

At no time during this research were VLSI interface constructs encountered that could not be represented in the CSP notation. That CSP should be equal to the task of representing these constructs is not surprising, because modern programming languages (such as APL, C, and Pascal) have been successfully used for this purpose, and CSP possesses all the expressive power of these languages in its sequential process description facility. Additionally, of course, CSP has a separate mechanism for scheduling and synchronization of these processes.

Two examples of CSP's power for specification follow. In the first, suppose one wishes to specify a sequence of actions when a signal SIG1 is high (1), another sequence when it is low (0), and a null sequence when it is undefined. CSP represents this specification as follows (in CSP/84 notation):

```
[ SIG1 == 1 → <statement_list_1>
| SIG1 == 0 → <statement_list_2>
| ~((SIG1==1)||(SIG1==0)) → skip;
                        /* Null access function */
]
```

If, on the other hand, if one wished to specify that an undefined value of SIG1 is not acceptable, he could have eliminated the alternative leading to "skip." This would cause the process to terminate on testing of an undefined value for SIG1. As a better expedient, the designer could "police" this condition by replacing "skip" with a trap to an error routine.

A second example illustrates the specification of timing constraints in CSP. The WASP 1.0 design [Hedl84a] contained a signal called "mgl" that was specified to rise and fall only on the leading edge of the "phi2gl" clock. This constraint can be represented in CSP simply by constructing the access function definitions in such a way that the specified behavior is obtained only if the access functions are called in the proper sequence:

```
*[ ...
| ? MGL = mgl → skip; /* Null access function */
| ? PHI2GL = phi2gl →
      <statements using new value of MGL>
 ]
```

Finally, CSP, or an equivalent notation possessing the power of CSP, seems a good compromise as a notation for abstract interface specification of VLSI designs. It is general enough to be reasonably portable and does not require any special extensions for use in IC design specification. However, it is more explicit than general-purpose programming languages (APL, C, Pascal, etc.) in expressing concurrency and in controlling complexity by effecting a clean partitioning of synchronization concerns from the expression of other aspects of module function. Furthermore, its ability to model subsets of the design as discrete processes makes it more suitable than general-purpose languages for evolutionary refinement as the design progresses.

9.9.1.2 Data/Control Mapping.

In the specification of IC designs using CSP, a certain convention for the mapping of data and control pins was useful. *Data* pins were best mapped using guarded CSP outputs and unguarded inputs, and *control* pins were best mapped using the reverse. This convention led to the definition of data pins by value access functions and of control by effect access functions.

One difficulty that can arise with such a convention is if a data output from one module is used as a control input in another module, because CSP forbids the connection of two guarded ports. A dummy module must be constructed and inserted between the two connections; this module consists only of an infinite loop consisting of an unguarded input and an unguarded output that passes along the result of the input. The introduction of such a dummy module is unclean, and it is not obvious that it will always produce an accurate specification. Further work is necessary here.

A second potential difficulty is deadlock, which can result from the connection of unguarded inputs and outputs (as would be the case if a control output was used as a data input). Middleton [Midd85] is modifying CSP/84, specifically for hardware modeling, to avoid this problem by preventing unguarded outputs from blocking: if an output is not accepted by the time a new output is generated, the first output is discarded and an error message is sent. In the specifications constructed in this research, I was always able to avoid deadlock, but as before such avoidance sometime involved the introduction of artifice (such as the forced "consumption" of an output value that would not be used). Such artifice also is present in other specification languages, and it does permit the accurate specification of the desired effect. Nevertheless, it is undesirable in that

it alters the correspondence between specification and implementation.

3.3.1.3 Access Function Extent.

A benefit of the proposed access function-based specification method is its potential for complexity control through crisp partitioning of module function into functional elements associated with each pin. In some modules, however, not much partitioning can be obtained. This difficulty occurs in particular in modules having many data pins and few control pins, because access functions for data pins tend to be trivial (value) access functions, leaving most of the module function remaining to be partitioned among the few control (effect) access functions. Although different abstractions for module state could probably be constructed to overcome this, such construction is not natural and would probably diminish the attractiveness of the method to designers.

When a better partitioning can be obtained, another problem arises. This problem is, where should the lines be drawn between the effects of the various access functions on the module state? For example, suppose there are two access functions S_mode and S_phi1 (a clock), in which the "mode" signal realigns the module environment for a different mode of operation. Should, on the invocation of S_mode, the entire module state be reset for the new mode in the current clock phase, or should the appropriate switches merely be set to activate the new mode when the next clock access function is invoked?

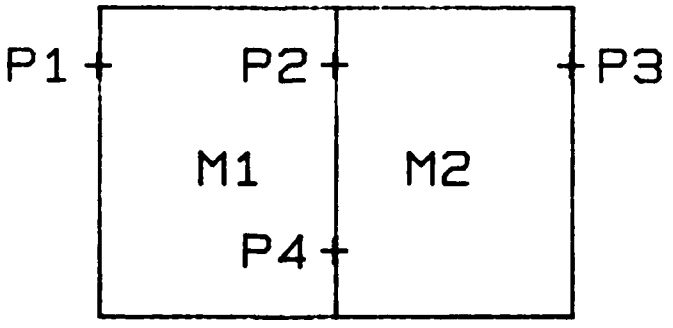
The answer to this question may differ depending on whether a specification or a simulator is being constructed. Recall that the goal of a specification is to provide a frame for verification, and thus it is required only to describe one model of correct operation and a catalog of specified actions to be taken when incorrect operation (as Parnas says, an "undesired event") is encountered. The requirements for a simulator are more stringent: a simulator should predict what will occur even in the presence of incorrect operation. It is important not to confuse these two sets of requirements, especially in that simulators can be used as specifications, as has been noted earlier. This is a pitfall of using a simulator as a specification: of the behavior included in the simulator, it is possible that the designer does not really know which behavior to implement (the "correct" behavior) and which to ignore.

3.3.1.4 CSP Base Language Support.

The concept of the CSP notation, as explained by Hoare [Hoar78], does not include a language for expressing the contents of the sequential process embedded in a guarded CSP command. In particular, the implementation of CSP I used, CSP/84 [Jaza80a, Jaza80b, Midd84], did not support closed subroutines in its base language. Even though escape to the programming language C was provided, there was no easy way that C subroutines could be linked into the CSP code. This omission was intentional, in that through a robust subroutine facility control transfers could be implemented that subvert the partitioning of scheduling that is the main attraction of CSP. Yet, for economy of coding, and particularly when greater levels of specification refinement with their more complex data types are being modeled, the abstraction that can be gained through closed subroutines is essential to understandability and cost-effectiveness of the specification. For the IC specification application, it is necessary that a CSP implementation include at least a restricted closed subroutine capability in its base language.

3.3.2 Performance.

Suppose two or more modules, each k_1-adhering to their specifications, are composed together. What, if anything, can be said about the k-adherence of the composition if adherence functions remain the same for pins that appear in both components and composition (Figure 3-8)?



M1 k_1-adheres to its specification; M2 k_2-adheres to its specification.

Figure 3-8. k-adherence of a Composition.

Before this question can be answered, the concept of "composition of specifications" must be defined: under what conditions is the composition of specifications itself a (consistent) specification, and what are the characteristics of such a composition? Let

$$M_1 = \langle P_1, S_1, V_1, W_1, VAF_1, WAF_1 \rangle \text{ and}$$
$$M_2 = \langle P_2, S_2, V_2, W_2, VAF_2, WAF_2 \rangle$$

be specifications. Then

$$M = \langle P, S, V, W, VAF, WAF \rangle$$

is a *composition* of M_1 and M_2 if the following conditions hold:

(1) P is a proper subset of $P_1 \cup P_2$;

(2) $S = S_1 \cup S_2$; and

(3) For each pin $p \in P_1 \cup P_2$ with associated access function $v(w) \in V_1 \cup V_2$ $(W_1 \cup W_2)$ and adherence function α $(\beta) \in VAF_1 \cup VAF_2$ $(WAF_1 \cup WAF_2)$, $p \in P \rightarrow v \in V$ $(w \in W)$ and $\alpha \in VAF$ $(\beta \in WAF)$. In other words, external pins in the composition retain their prior access and adherence functions.

The composition is called *consistent* if the difference $(P_1 \cup P_2) - P$ is a set of pairs of pins

$$\{ \langle p_1, p_2 \rangle \mid p_1 \in P_1, p_2 \in P_2 \}$$

such that

(a) Exactly one pin of each pair has an associated access function defined (rather than called) in $V_1 \cup V_2 \cup W_1 \cup W_2$; and

(b) The access functions corresponding to the pins in the pair are neither both value nor both effect access functions.

In a real sense, then, the composition behaves as the sum of its component parts, each retaining its identity and behavior with no real concern for its environment. Such information hiding, as has been noted, was a goal of this approach for change management. The consistency condition is really a syntactic check, akin to the verification that inputs and outputs are connected in pairs, and is not central to the development. Note that mixed-level composition is also permitted by this definition; as section 3.1.2.1 indicates, such composition requires only the presence of appropriate data type conversion facilities.

Assume, then, as in Figure 3-8, that two modules M1 and M2 are composed, with adherences k_1 and k_2, respectively. Observe that it is possible for the composition to exhibit k-adherence for k greater than either k_1 or k_2; for suppose pin P1 and P3 both 1-adhere to their specifications, pin P2 k_1-adheres to the specification of M1, and pin P4 k_2-adheres to the specification of M2. In this case it would be possible for the composition to 1-adhere to its specification even though k_1 and k_2 were substantially less than 1. Shortfalls in the adherences of P2 and P4 can be completely compensated for by large safety margins built into their corresponding pins in the adjacent module. A composition, then can exhibit 1-adherence even though neither component does.

What, on the other hand, is the worst case? Suppose, in Figure 3-8, that all pins of M1 k_1-adhere and all pins of M2 k_2-adhere, $0 < k_1, k_2 < 1$. Is it then possible that the composition might exhibit k-adherence for $k < \min(k_1,k_2)$? A moment's reflection will show that such a case is also possible. Suppose all pins but P2 1-adhere to their respective specifications, and P2 0.5-adheres to the specifications of both M1 and M2. Then, by definition, both M1 and M2 0.5-adhere to their specifications, but if the diminished output from M1 at P2 is insufficient to drive the higher-than-expected load in M2 at P2, M2's external output pins might not function at all, and the composition would exhibit 0-adherence.

Given simply that two modules k_1-adhere and k_2-adhere, then, the level of adherence of their composition cannot be predicted *a priori*. It would be helpful to conduct further research in this area to determine the kinds of restrictions on design parameters that would permit such a prediction to be made. Udding [Uddi84] has recently published some work of this kind, which provides tests for the robustness of module delay-insensitivity through composition; additional work is ongoing [Moln85].

As an example of the type of results that are needed, observe that a lower bound on composition adherence can be obtained if adherence shortfalls in one module can be "translated" into adherence shortfalls in the other. Suppose, in Figure 3-8, that pin Pi k_{ij}-adheres to module specification Mj. Suppose further that signals flow through both P2 and P4 from M1 into M2. Finally, suppose the capability is available to "translate" the adherence shortfalls $(1 - k_{21})$ and $(1 - k_{41})$ by decreasing k_{22} and k_{42} (to, say, k'_{22} and k'_{42}) enough to raise k_{21} and k_{41} to 1. That is, pin pairs with the adherences (k_{21}, k_{22}) and $(1, k'_{22})$ would behave identically; so would pin pairs with (k_{41}, k_{42}) and $(1, k'_{42})$. In this restricted case, then, the composition adherence is bounded

below by

$$\min [k_{11}, k'_{22}, k'_{42}, k_{82}].$$

8.4 Simplifications to the Approach Permitted by VLSI Design Characteristics.

As has been shown, the abstract interface specification method proposed in the preceding sections has many desirable attributes; however, it is also costly to build and use. Much of the cost comes from the method's generality. In this section I will examine ways of exploiting characteristics of VLSI design to reduce this cost.

In general, cost reduction can be pursued through restriction of the method to certain standard formats that are of interest to VLSI designers. First, for example, a standard language can be used for expressing access functions, data type definitions, and access function definitions. A standard language improves executability characteristics by capitalizing on portable, optimized system software developed for interpreting this language. Also, it improves intellectual control by allowing designers to express function with familiar programming objects. In this research I chose to use the language C as such a standard, chiefly because my CSP interpreter, CSP/84, was C-compatible.

Semantic Refinement:

1. Functional: Three values (0,1,X); no signal strength information.
2. Four-strength: Three functional values plus four signal strength levels (unknown, floating, steered, restored).
3. Drive-load: One voltage level functional value plus an average switching current value. Rise and fall times are symmetric.
4. E11: Voltage levels plus switching current as reflected in nominal pin voltages, resistances, capacitances. Rise and fall times may be asymmetric.

Syntactic Refinement:

1. Immaterial: No geometric information.
2. Pingrid: (X,Y) virtual grid coordinates provided for each pin.

Table 3-2. Specification Levels Used in this Research.

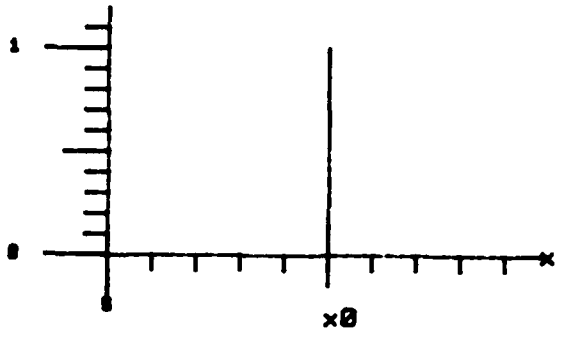
Next, one can restrict the specification refinement hierarchy of Figure 2-4 to certain levels at which generic state types have been predefined. In this way, each specification need not contain full definitions of these data types; instead, it need only refer to previously catalogued definitions. The specification refinement level is entirely a designer option, reflecting his/her evaluation of the cost/effectiveness tradeoffs inherent in constructing specifications at such a level. The introduction of a new level involves only the definition of a new generic state type and a corresponding set of adherence functions. Therefore, I make no claim that the types I have chosen are *the* types that should be used in specification construction. Table 3-2 lists the specification levels that were used in this research; some of their definitions have been included in the specifications in Appendix A.

Together with a standard generic state type set, one can construct a matching set of standard adherence functions that operate on data types at each refinement level. Table 3-3 contains definitions of the standard adherence functions that were chosen for this research.

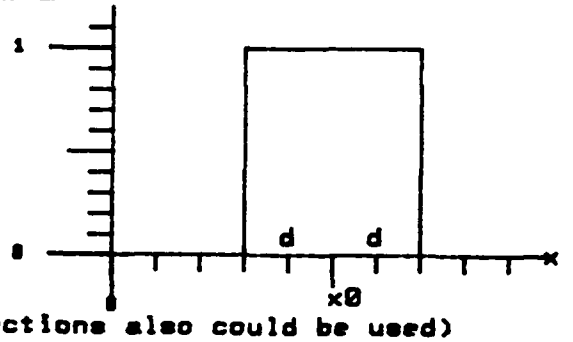
Why were the functions in Table 3-3 chosen? The first function, labeled "Functional" or "Spike," is the implicit function inherent in all functional specifications. The others are generalizations of this function that incorporate various kinds of specification tolerances. The Square Band function is a direct generalization of the first function, but it is insensitive to changes in k when k-adherence is being measured. Therefore, the Linear Ramp function, which provides the same tolerances as the Square Band but which is linearly sensitive to changes in k, was introduced. The Normal function has better mathematical properties for composition (e.g., continuous derivatives) than the other functions and is also sensitive to changes in k; it was considered to investigate the utility of these better properties.

Finally, when such variables as signal strength are being considered, one-sided functions are of more use than two-sided; signal drive greater than some minimum, or signal load less than some maximum, is generally of no concern. Consequently, one-sided alternatives were also considered for all adherence functions studied. Of course, an adherence function can be a combination of two-sided and one-sided functions; in general, it may be any combination of functions on its data type components. For example, Figure 3-6 contains for the "P_" access functions a combination of Spike and One-Sided Square Band adherence functions: the Spike function measures adherence on the data type component *sigfval*, while the One-Sided Square Band function measures adherence on the data type component *sigstrength*.

1. Functional (spike)

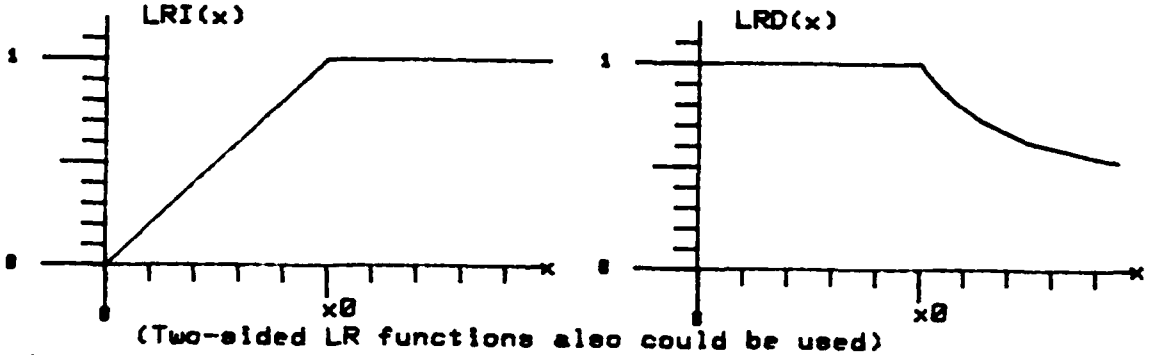


2. Two-sided square band



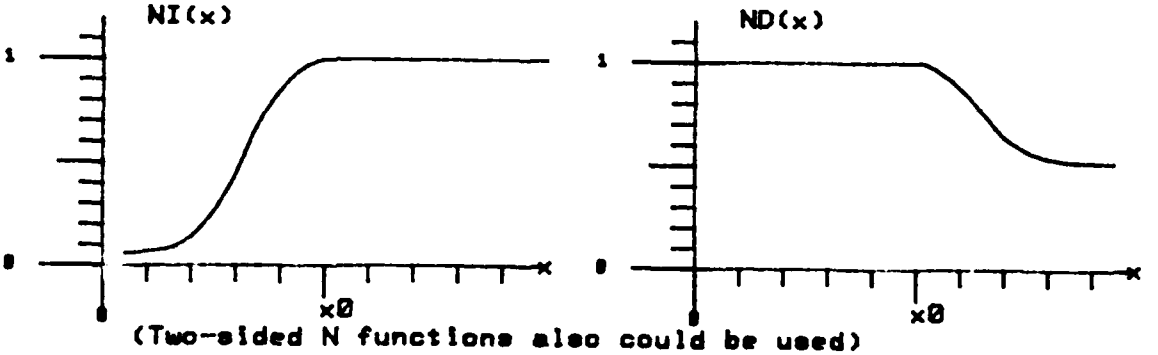
(One-sided square band functions also could be used)

3. One-sided Linear Ramp (LR)



(Two-sided LR functions also could be used)

4. One-sided Normal (N)



(Two-sided N functions also could be used)

Table 3-3. Adherence Functions Used in this Research.

In summary, compared to Figure 3-6, the incorporation of these simplifications reduces the size of an interface specification constructed with the proposed method. Nevertheless, the cost of a good specification is high, and it increases with the level of specification refinement. This cost can be justified only by comparing it to the cost of a bad specification. Because there have been no controlled studies that quantify this comparison (nor are there likely to be, because of the cost and difficulty of such studies), each designer must decide to what extent his/her individual project merits the use of interface specifications. The object of this research is to show that, while good abstract interface specifications for VLSI-scale design are complex, the application of software technology can reduce their life-cycle cost. In the following chapter I will describe experiments which suggest how well the proposed approach attains this objective.

CHAPTER 4

TECHNIQUE EFFECTIVENESS

In chapter 3, I reported on the development of a new approach to VLSI design abstract interface specification, illustrating the approach with a method that employed it. This chapter contains an evaluation of the approach using the criteria laid down in section 2.2 for abstract interface specification techniques. Four separate categories of questions are addressed in this evaluation:

- How well does the method scale up to VLSI complexity levels, and is it perceptibly cost-effective at these levels?
- How easily can specifications be refined? Is the method malleable and adequate throughout?
- Is the method spare? In particular, can geometric information be included with the same mechanisms used for functional and electrical information?
- How well does the specification manage change? Does the specification approach inhibit the propagation of change to other parts of the design?

Experiments or studies were conducted to gain insight in each of these areas. In two subsections, this chapter describes first the experimental design and then the experimental results.

4.1 Experimental Design.

4.1.1 Scalability Test.

Unlike clear-box specifications, which control complexity with hierarchy, black-box specifications use abstraction as their primary complexity management technique. That suitable abstractions exist for VLSI-scale modules is not in question: examples of such abstractions are many and varied. It remains only to show that abstractions of large modules can be effectively used in the proposed class of methods.

The only obstacle to such a demonstration is the traditional one: the unavailability for study of detailed public-domain VLSI designs. Such designs are sufficiently costly to develop that they are beyond the threshold of pure research laboratories, and the designs developed for the marketplace remain proprietary. Rather than construct an artificial example (even if the

resources for such construction were available), I chose to specify as demonstrations of the technique SSI, MSI, and LSI designs available locally. The extrapolation of complexity factors present in these designs should provide insight into the scale and perceived cost-effectiveness of specifications at VLSI levels.

The proposed method was used to construct abstract interface specifications of several smaller designs as well as a specification of an LSI chip, the WAfer-Scale Systolic Processor (WASP) developed in 1984 by Kye Hedlund and his associates [Hedl83, Hedl84b, Cole85]. Based on the relative costs of constructing these specifications, estimates were made of the scalability and perceived cost-effectiveness of the method. These estimates are presented in section 4.2.1.

A word about the selection of designs for this testing is in order. These designs, whose specifications are shown in Appendix A, are all real designs, originally implemented in full-custom nMOS combinational and sequential logic. To keep the experiments manageable, the size of the experimental test set was kept small; therefore, certain types of designs were not included. For example, I deferred consideration of semi-custom modules for later research; however, because semi-custom design is more constrained than full custom design, it seems intuitive that a system that manages complexity satisfactorily in the full-custom design environment should translate well to the semi-custom environment. Also, because the specification method offers clock signals no special semantic interpretation, I made no effort to include modules that were specifically intended for asynchronous operation. Finally, because the method encompasses analog specification in the course of refinement, I included no modules specifically intended for analog implementation.

The design modules selected for this specification testing are summarized in Table 4-1.

4.1.2 Malleability Test.

Whether or not a specification is malleable depends on (1) its ease of being refined with detailed design parameters as these become available; and (2) its service to the designer at many points in the life-cycle. The latter question may be rephrased, "Does the specification continue to provide a frame for verification at numerous design life-cycle points?"

Module Name	Size	Description
plashin	7	nMOS shift register cell (sequential logic). This module was evaluated in the context of a shift register of four such cells.
fnblk	8	nMOS combinational logic. This module was evaluated in the context of the 'malu' (next entry).
malu	35	nMOS microprocessor ALU; composition of 5 smaller cells. This module was evaluated in the context of the microprocessor datapath.
wasp	3860	Array of simple processors embedded in a switch matrix. Processors can perform the 16 Boolean functions on two bits.

Table 4-1. Experimental Module Test Set.

Source of designs: WASP [Hedl84a, Hedl84b].
Sizes are in numbers of transistors in an actual implementation.

To gain a feeling for how easy it would be to refine specifications written using the proposed approach, I refined a design through three different semantic levels and two different syntactic levels. I used the number of specification lines changed in each refinement as a rough measure of refinement effort. These results are presented in section 4.2.2.

Next, I executed the specifications thus constructed at each semantic refinement level and evaluated informally the ease of comparing these outputs to simulation output from actual designs at each level. I further examined the ease of verifying syntactic specification components with actual design syntax at each syntactic refinement level. The results of these evaluations are also presented in section 4.2.2.

4.1.3 Spareness Study.

To evaluate the method's spareness, each specification element (enumerated in section 3.2) was judged informally according to the following criteria:

(1) Is the information included in this section essential to design? What part, if any, of the design could not be accomplished if information in this section were omitted?
(2) Is the information included in this section redundant, that is, presented anywhere else in the specification? Could the information in this section be presented more compactly?

The results of this analysis are presented in section 4.2.3.

4.1.4 Change Management Tests.

The proposed method's effectiveness in change management was evaluated in three ways:

(1) *(Vertical Change Propagation).* First, I changed the specification of a module that had already been decomposed and had had its components specified. I measured the amount of change that propagated to the component specifications, using once again the metric of number of specification lines changed.
(2) *(Horizontal Change Propagation).* Second, I investigated the conditions under which less than full adherence to a module specification affected that module's environment. The horizontal change propagation test consisted of the following steps:
   - I developed a representative sample set of design modules, each in the context of a larger design.
   - I simulated the performance of the modules in context.
   - I made "typical" changes to the modules and evaluated the k-adherence of the changed modules to the specification.
   - I re-simulated the performance of the changed modules in their context.
   - I determined under what conditions, if any, change propagated to the studied modules' environments.
(3) *(Internal Robustness).* Third, I measured the amount of modification that was needed in a large specification to reflect several actual changes the design had undergone.

4.2 Experimental Results.

4.2.1 Scalability Test.

The goal of this test was to deduce, using data from the experimental specifications, an idea of how large the specification for VLSI-scale designs would be.

Table 4-2 summarizes the experimental data, and Figure 4-1 shows the relationships between design sizes (in number of transistors) and specification sizes (in lines). The points labeled "F" represent the absolute size of specifications constructed at the functional and four-strength refinement levels; The points labeled "D" represent the absolute size of specifications constructed at the drive-load refinement level; and the points labeled "I" represent the size of the former specifications with their level-dependent sections removed.

These data cannot, of course, be used to guarantee the practical scalability of the proposed approach. They are presented merely to show that, for the specifications considered, specification size grew slowly with design size.

4.2.2 Malleability Test.

As was explained in section 4.1.2, the evaluation of the proposed approach's malleability consisted of two phases. In the first phase, I found that specification refinement permitted the following percentages of specification lines to remain unchanged:

Refinement (Table 3-2)	Lines Unchanged[1]	% Unchanged
functional to four-strength	77/82	93.9
four-strength to drive-load	69/161	42.9
functional to drive-load	69/161	42.9
immaterial to pingrid	82/112	73.2

[1] Notation: Entries are of the form n/d, where n is the number of specification lines unchanged in the indicated refinement, and d is the larger of (total number of lines in the unrefined specification, total number of lines in the refined specification).

Spec'n Number (App.A)	Spec'n Levels (sem/synt)	Transistors (typical)	Specification Lines Level-Indept	Level-Dependt	Total
B-6,7	fnal/immat	7	36	41	77
3-6	4str/immat	7	36	46	82
A-1	drvl/immat	7	36	125	161
A-2	4str/pingr	7	36	76	112
A-3	drvl/immat	8	28	151	179
A-4	drvl/immat	35	87	191	278
A-5	fnal/immat	3860	113	134	247

Table 4-2. Sizes of Experimental Specifications.

Note: "Specification lines" figures count expansions of closed-form adherence functions (because parameters may differ from pin to pin), iterative definitions (such as "F[B], B=0..3") as one line, and closed forms of open subroutines. Comments, blank lines, headers, the "Assumptions" section, and code used for consistency checking only were not counted.

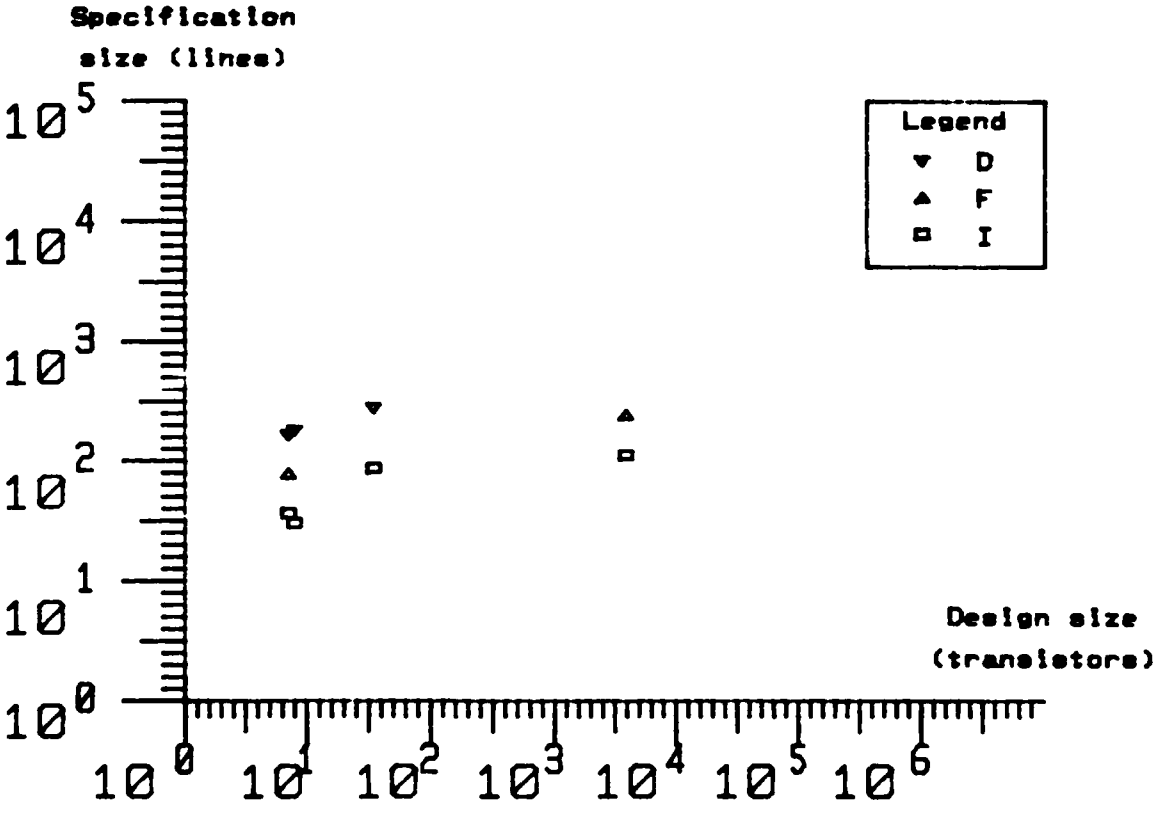


Figure 4-1. Scalability of Proposed Specification Method.
(See text for legend.)

This analysis confirmed that the partitioning inherent in the specification method allowed a substantial fraction of old specifications to be used in refined specifications. Also, since the changed portion consisted largely of data type and adherence function definitions, which can be reusable, the potential exists for making refinements with little effort indeed.

Further, although the module studied in the previous analysis was a small one, it seems realistic to believe that this good performance will hold in the specification of larger modules as well, in that the percentage of specification lines that was refinement-level-dependent remained roughly constant regardless of module size:

Spec'n (Fig.)	Transistors (typical)	% of Specification Level-Dependent Functional	4-Strength	Drive/Load
B-7	7	53.2		
3-6	7		56.0	
A-1	7			77.6
A-2	7		67.9	
A-3	8			84.4
A-4	35			68.7
A-5	3860	54.3		

Furthermore, the complexity of the transformations carried out in specification refinement appears to be independent of module size. Although larger modules require the definition of more different data type operations, this increased volume is reflected in specifications at all refinement levels. In summary, the specifications studied in this research exhibited good reusability and thus were easy to refine.

Using the specification to verify the design at the functional and four-strength refinement levels was straightforward. The output from both simulation and specification interpretation was in the same digital format, and, because spike adherence functions could be used meaningfully at these levels, comparison of the two sets of outputs for equality satisfied the verification task. Verification decisions were rendered as either 1-adherence (if all module pins adhered fully), or (otherwise) 0-adherence. At the pingrid syntactic level, spike adherence functions are no longer useful in general, but this simply meant that verification decisions could be rendered in terms of k-adherence for all k in [0,1].

At the drive-load semantic level, an inconvenience arose: the output of the simulator used was in a format (graphic waveforms) different from the output of the specification interpreter (numeric text). While this problem was not a conceptual difficulty preventing effective verification of the design, it does illustrate the desirability of integrating such a specification interpreter into the design tool suite.

4.2.3 Spareness Study.

The proposed method yields a specification with seven sections or elements. The first three sections are informational, providing an immediate abstract view of the module to the designer. The next two sections are more detailed, and the final two more detailed still. Through its partitioned structure, the proposed specification approach presents the module in a layered way and identifies which details should be included and which omitted.

The following observations may be made about the essentiality and redundancy of the information contained in each of the seven elements of the proposed specification:

(1) *Assumptions.* As was the corresponding section in Britton, Parker, and Parnas's software specification [Brit81a], this section is "partially redundant." In addition to its casting important portions of the formal sections to follow in a more easily-understood way, it also explicitly states premises that underlie the construction of the remainder of the specification. It would be difficult to justify the essentiality of this section, but without it the specification would communicate much less effectively. The assumptions section, then, is included for efficiency rather than for completeness.

(2) *Specification Level.* This small section identifies the specification data types being used. Like the assumptions section, it too is informational.

(3) *Pin and Internal State Variable Summaries.* The information in this section essential to the specification process also could be inferred, in this case from the access function section. Nevertheless, this index facilitates the construction of a specification interpreter and also provides an abstract view of the module.

(4) *State Variable Attribute Initializations.* This section, which directly follows the summaries of pin and internal state variables, declares particular values for their level-specific attributes. Consequently, this section is clearly essential, in that it directly provides syntactic reference values for static verification. Further, semantic values in this section are referenced by access functions for dynamic verification. The method provides identical facilities for including both semantic information (usually functional/electrical) and syntactic information (which tends to be geometric).

(5) *Access Function Definitions.* The heart of the specification, this section specifies the behavior of each IC module pin. The partitioning of module function into per-pin components, even when these components are identical or similar, need not produce excessive descriptive redundancy, because common elements can be factored out and described by a reusable data type definition. The treatment of timing through a uniform implicit discipline does make the description of function slightly less compact, but large compensating dividends are gained by the separation of concerns this partitioning produces.

(6) *Data Type Definitions.* The data structures, and operations on them, comprising the generic state type are defined in this essential section. Separation of level-specific definition into this section from the access function definition section is less than optimally compact; as before, however, the ease of refinement that such a separation imparts, with the corresponding gains in malleability, compensate for this type of presentation.

(7) *Adherence Function Definitions.* Without this section, the specification could not convey a precise understanding of the relative merits of various degrees of adherence. Further research is required to determine whether this concept can be conveyed more concisely than with such functions.

4.2.4 Change Management Tests.

4.2.4.1 Vertical Change Propagation.

In this test, I measured the amount of change propagated through a single decomposition level by several real-world changes applied to a parent specification. The changes applied are enumerated in Table 4-3.

The results of this test are reported in Table 4-4. Usually, the propagated complexity was of the order of the original change, a pleasant result considering that several types of changes and differing specification refinement levels were considered. For the changes numbered 2 and 4, however, there was an order of magnitude increase in the propagated change.

Change No.	Description
1	Add a pin to clear the accumulator in each of the nine WASP processing elements.
2	Generate an internal control signal to recirculate the internal state of the processing elements.
3	Correct an error in the logic of the processing element function block.
4	Change the edge membership of module pads to effect a more balanced distribution around the chip perimeter.
5	Change from a broadcast to a pipelined clocking discipline.
6	Change scan-in/scan-out lines from driven to precharged.

Table 4-3. Types of Changes Studied for Vertical Propagation Effects.

Change No.	Semantic/ Syntactic Refinement Level (Table 3-2)	Spec'n Lines Changed/ Total Lines in Parent	Spec'n Lines Changed/Total Lines in First-Level Children	Percent Propa-gated
1	funct/immat	5/247	5/483	100
1	drvload/immat	7/308	7/278	100
2	funct/immat	2/247	22/503	1100
3	funct/immat	4/247	4/503	100
3	drvload/immat	16/278	16/775	100
4	funct/pingrid	98/443	887/1504	905
5	funct/immat	19/247	57/475	300
6	funct/immat	0/247	0/503	0

Table 4-4. Vertical Change Propagation.

Notation: Columns 3 and 4 present a ratio, n/d. In column 3, n is the number of lines changed in the parent specification and d is the larger of (total number of lines in unchanged parent specification, total number of lines in changed parent specification). In column 4, n is the total number of lines changed in all the specifications for the first-level child modules of the parent, and d is the larger of (total number of lines in all unchanged first-level child specifications, total number of lines in all changed first-level child specifications).

— In the first of these cases, change 2, a new internal module had to be specified: it was the fixed cost of constructing an entirely new submodule specification (even though only 20 lines long) that caused the large percentage change. This result suggests that, while in-place specifications may be easy to change, there is a disproportionately high start-up cost for new

specifications. Tools that assist in creating routine portions of the specification could alleviate this cost.

— The second case, change 4, exhibited poor robustness chiefly because the proposed specification approach itself was insufficiently applied. The syntactic data type chosen, *pingrid*, called for the specification of absolute coordinate points on a two-dimensional grid. This was not a good use of abstraction. Consequently, each change to the parent module specification rippled through several first-level child modules. A better abstraction, for example one in which edge membership is described symbolically, could be expected to resist change propagation more fully. A corollary observation might be that less abstract data types appear more susceptible to vertical change propagation.

4.2.4.2 Horizontal Change Propagation.

The types of test changes to be introduced into the test module depended on two factors. First, there was the level of refinement of the module specification. The signal data types at the functional specification level are not rich enough to admit value perturbations and still have their modules k-adhere to specifications for any $k>0$. Slight specification refinement (to include, say, a small finite set of signal strengths), produced similarly unsatisfactory test-bed specifications. I chose, therefore, to specify test modules at a more refined level. Generally, the drive-load functional/electrical level (Table 3-2) was used: it was simple, but not so much so that essential specification elements were not illustrated.

The choice of adherence function also influenced the types of changes that would be found interesting. Spike adherence functions (Table 3-3), which reduce k-adherence to 1-adherence for all $k>0$, and Square Band adherence functions, which are also insensitive to change from a k-adherence standpoint, were discarded for analysis. The Linear Ramp and Normal functions, therefore, were chosen for the specifications used in this study. For consistency of analysis, I used the one-sided adherence functions illustrated in Figure 4-2: the decreasing functions are reciprocals of the increasing Linear Ramp and Normal functions, so that a zero drive and an infinite load both have 0-adherence. Once the specification, including adherence functions $\alpha(x)$, was established, changes were introduced by perturbing output values to those values of x for which $0 < \alpha(x) < 1$.

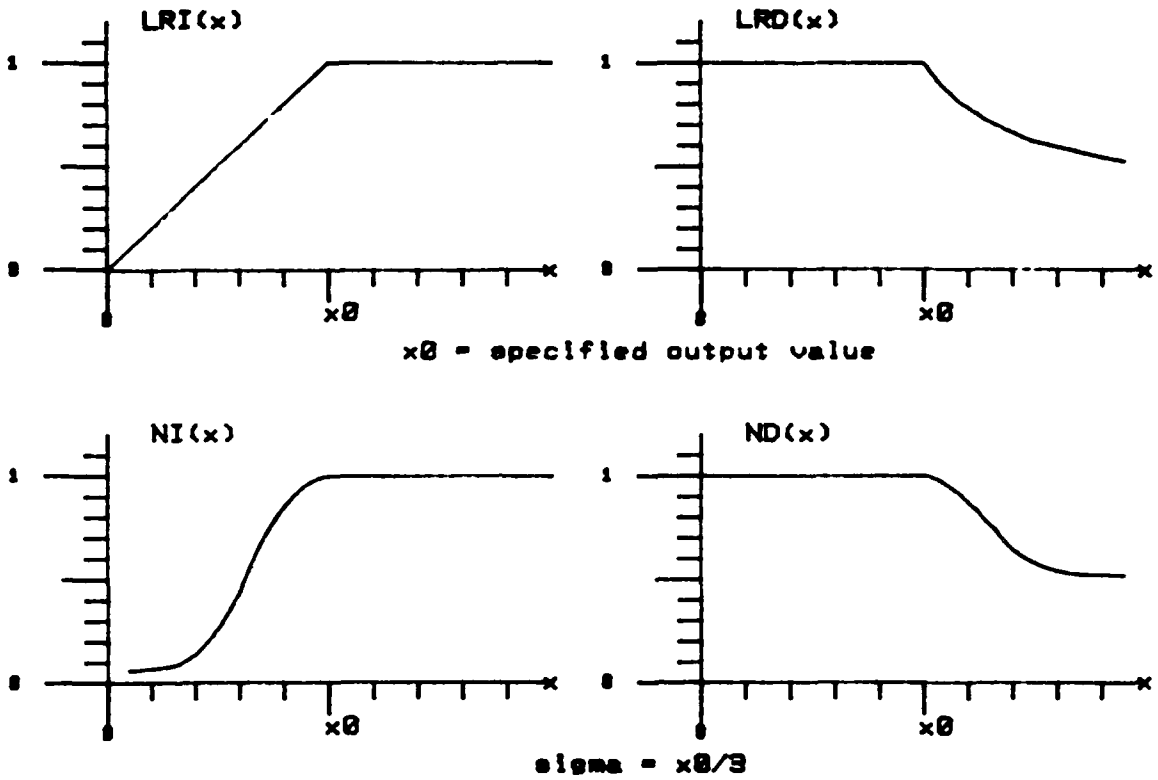


Figure 4-2. Linear Ramp (LR) and Normal (N)
Increasing and Decreasing (I/D) One-Sided Adherence Functions
Used in this Research.

Figure 4-3 illustrates the canonical test environment. It consists of an incoming signal path, a pin on a module boundary, a propagation path, and an observation point. Note that the observation point can be outside the test module, in which case the pin is an output pin and the propagation path in the module's environment, or it can be inside the test module, in which case the pin is an input pin and the propagation path inside the test module.

The testing showed that quantifying the amount of propagated change depends on knowing both the k-adherence of the test module and the characteristics of the propagation path. Alternatively, the test suggested that different adherence functions should be used depending on the intended utilization of module pin signals, so that "k-adherence" acquires a uniform meaning for each value of k.

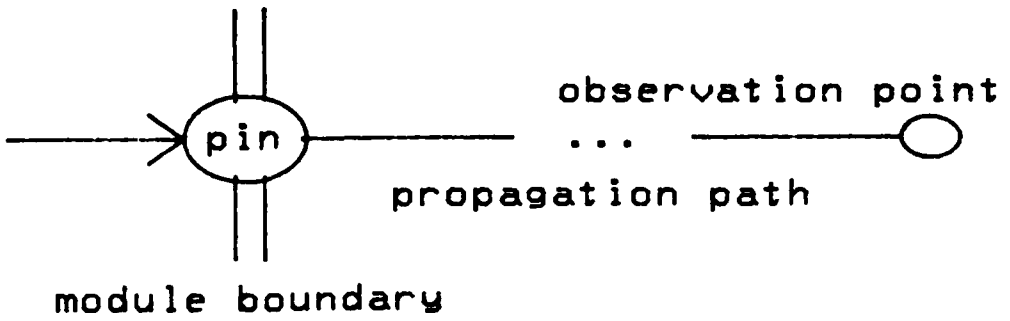


Figure 4-3. Canonical Test Environment.

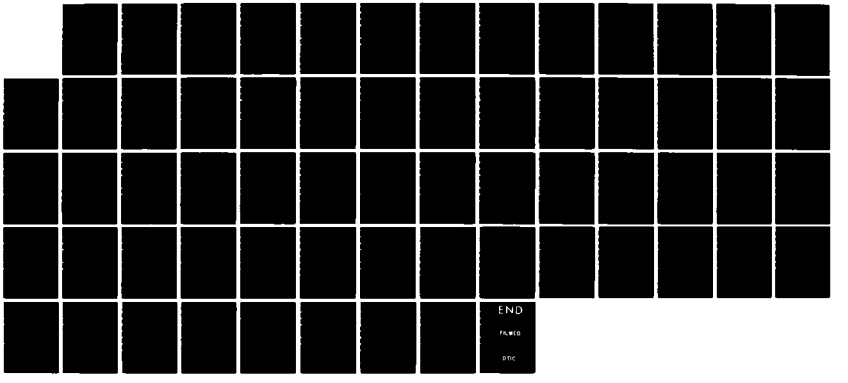
For small discrepancies in adherence, a more complex propagation path tended to ameliorate the effects of less-than-full adherence to a module specification. The simplest type of path measured was a bus (Figure 4-4). Electrically, the pin and observation point are the same node in this arrangement, so that the waveforms observed at the two points are identical. Less than full adherence to the specification at the pin, therefore, translates directly into a similar deficiency in the environment. The implication is that it is important for long (e.g., control) signal drivers to adhere fully to their specifications; consequently, adherence functions for bus drivers should be steeply sloped.

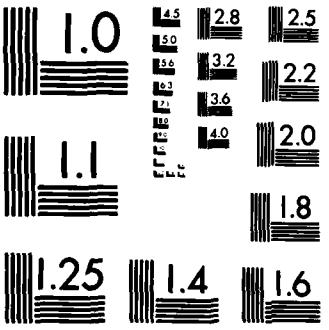
Even a simple change to the propagation path reduces the effect of less-than-full adherence. Figure 4-5 illustrates change propagation through a steered signal path in nMOS. Here the pin and observed waveforms, originally separated by a threshold voltage drop, tend to come together as k diminishes.

This effect is even more pronounced when signal restoration is included in the propagation path. Figure 4-6 illustrates that severe degradation of the pin signal is possible before an effect is noticed at the observation point. However, as Figure 4-6 indicates, a rapid crossover occurs at low values of k: suddenly the observed signal becomes considerably worse than the pin signal. This phenomenon indicates that, for more complex propagation paths (especially those including restoration of signals), careful design in the propagation path may achieve moderate tolerance to less-than-full specification adherence. The adherence function for such pins, then, should be gently sloped around the specified value, with a steep slope being attained around the point of behavioral failure. Table 4-5 summarizes these results.

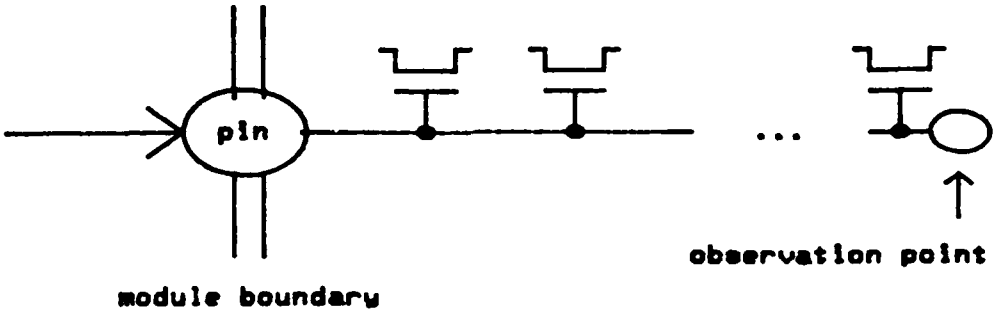
AD-A158 034 USING SOFTWARE TECHNOLOGY TO SPECIFY ABSTRACT INTERFACES IN VLSI DESIGN(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH R R GROSS 1985 2/2

UNCLASSIFIED AFIT/CI/NR-85-92D F/G 9/5 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A



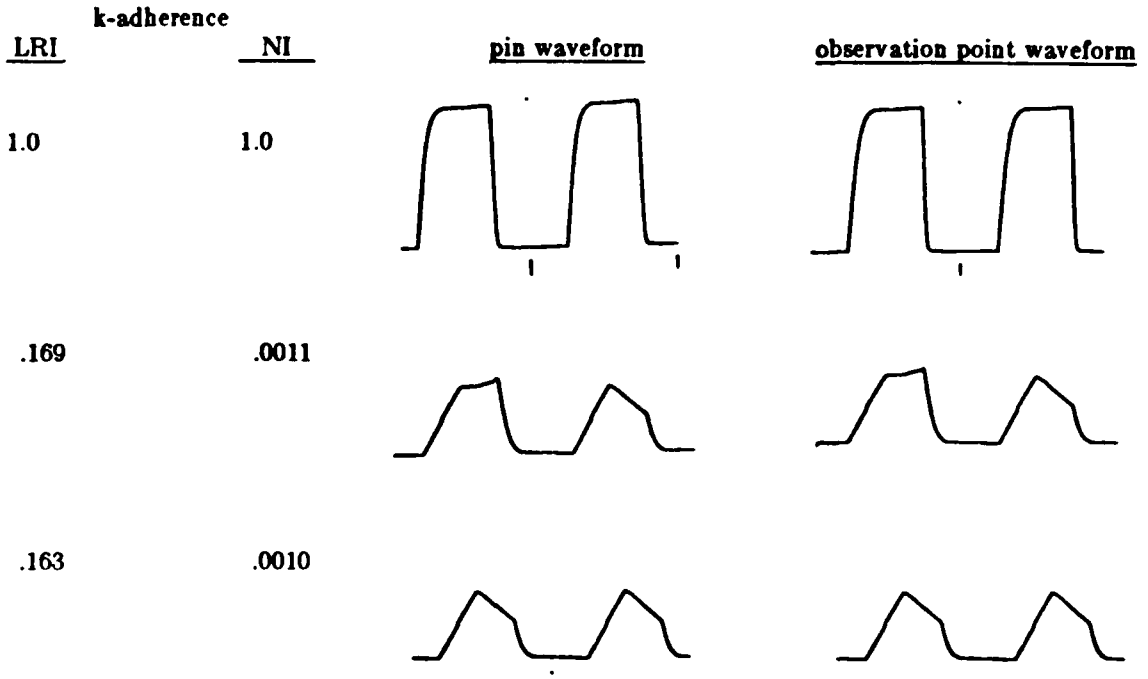
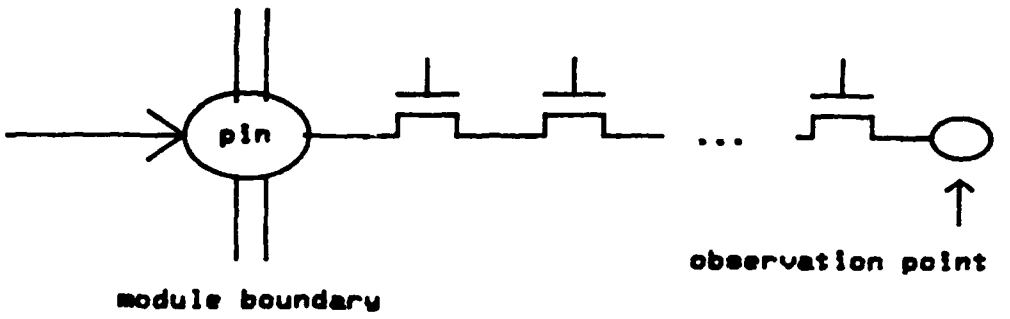
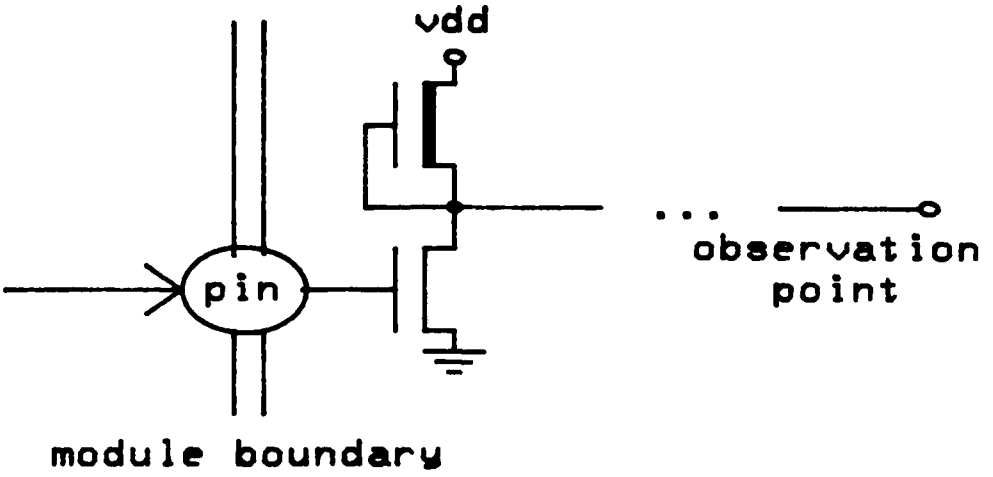


Figure 4-4. Change Propagation through a Bus.



k-adherence LRI	NI	pin waveform	observation point waveform
1.0	1.0		
.169	.0011		

Figure 4-5.
Change Propagation through a Steered Signal Path in nMOS.



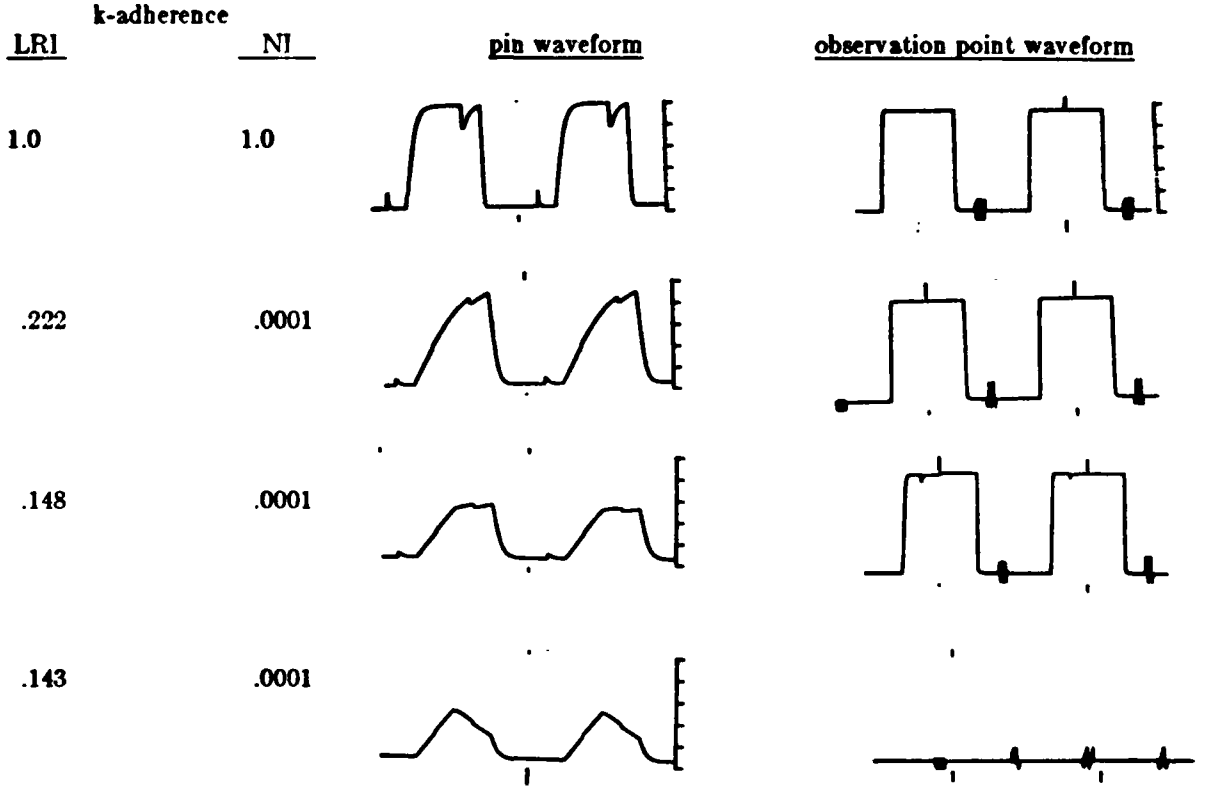


Figure 4-6.
Change Propagation through a Restored Signal Path.

It is not evident from these data that continuous adherence function curves are needed. Piecewise linear functions should suffice.

Type of Pin	Adherence Function Characteristics
bus driver	should fall off sharply when specified value not achieved
pass-gate logic	near-unit slope probably ok for nMOS implementations
restored logic	tolerant to small discrepancies; should fall off sharply around failure point for large discrepancies

Table 4-5. Desirable Adherence Functions for Controlling Horizontal Change Propagation.

4.2.4.3 Internal Robustness.

Having illustrated how well the proposed approach resists change propagation from module to module, I now will examine how much change is introduced into a single module specification by typical real-world changes. Each of the changes enumerated in Table 4-3 was an actual change made in a real design. Observe from the statistics for parent modules in Table 4-4 that each of the changes, with one exception, resulted in the modification of less than 10% of the lines in the specification. Because this specification approach emphasizes partitioning and minimal redundancy, it is not surprising that the studied changes did not result in major specification rewriting.

The one change (change 4) requiring over 10% of the specification lines to be changed has been identified before as an instance of a less-than-optimal usage of the proposed approach. That is, in this example insufficient use was made of abstraction. A more abstract data type for the syntactic specification information, such as an extension of the abstractions used in modern symbolic layout systems (e.g., VIVID [Rose84]), would have improved specification robustness in this example. With this specification approach, as with all representation systems, it is true that robustness under change declines as the representation data types become more concrete. Yet even with the rather concrete data type used in this example, the partitioning inherent in the

proposed approach prevented the need for wholesale specification changes: the worst-case change of 22% is certainly tolerable.

CHAPTER 5

CONCLUSION

In this final chapter, I will summarize the principal conclusions from this study and make suggestions for future research.

5.1 Principal Conclusions.

5.1.1 Software Technology Transfer to VLSI Design has been Demonstrated.

Despite its intuitive appeal (cf. chapter 1), the transfer of software technology to VLSI design has as yet been a largely unfulfilled promise. Although similar approaches to common problems can be observed in software engineering and VLSI design, few of these similarities can be traced to an explicit attempt to transfer software technology [Gros83c]. As a result, some skepticism has developed of late, and researchers are now seeking demonstrations that confirm the promise of software technology transfer.

The research carried out in this dissertation is such a demonstration. Abstract interface specification techniques developed exclusively in the software domain were extended and applied successfully to VLSI design. To recapitulate, these techniques included: (1) using the definitions of abstract data types in interface specifications; (2) obtaining the precise partitioning and complexity control of module interface specifications using a finite state machine and access function model; and (3) expressing and interpreting these interface specifications with Hoare's Communicating Sequential Processes (CSP) notation. That the transfer effort was successful is evidenced by the demonstrated realization in VLSI interface specifications (chapter 4) of the benefits of these software techniques.

5.1.2 The Proposed Abstract Interface Specification Approach Has VLSI-Level Power.

To repeat from chapter 1, VLSI design is integrated circuit design in which brute force no longer works. This research has proposed an approach for abstract interface specification that demonstrates specification power appropriate to VLSI complexity levels.

Reasonable extrapolations of parameters from specifications constructed in this study suggest strongly that the proposed approach can be used to specify functionally in tens of pages abstract interfaces of designs whose implementation requires tens of thousands of devices. Furthermore, both functional and performance specification are supported at these levels of integration, and high-level specifications can be evolved (with non-trivial reusability of their contents) to provide a verification frame at later design stages. A third implication of the data in this study is that the total size of performance specifications, if needed, is manageable, roughly an order of magnitude larger than the size of the corresponding functional specification. A method of quantifying adherence to the specification is provided for use at these levels of detail. In summary, the research provides a means of realizing and exploiting abstraction and precise partitioning for VLSI-scale complexity control.

5.1.3 VLSI Design Specification is Difficult and Not Universally Inspiring.

This research also confirmed that precise specification, a difficult task in the software domain, is no easier in VLSI. Numerous classes of component assumptions (section 3.2) must be included if the interface specification is to be complete. Identification and inclusion of these assumptions is a resource-consuming process; abstract interface specifications take longer than expected to construct and are much larger than clear-box specifications now commonly used in VLSI design. As has been noted, of course, such clear-box specifications use a monolithic (unpartitioned) approach that strains or exceeds designers' intellectual capacities, even at MSI/LSI levels.

In this regard, I should note that I found in the process of conducting this research that the value of specifications *per se* in VLSI design is not yet commonly embraced[1] (a difficulty that also exists in software). Accordingly, much of the research commonly suggested in the following section is directed toward demonstrating this value. Without a greater base of agreement on the merits of VLSI specifications, research into this topic will be difficult to support.

---

[1] For example, an anonymous referee wrote, in response to an earlier partial draft on this research, "As a designer I hate formal description languages and the idea of executable specifications. I think both interfere with my ability to design effectively."

5.2 Suggestions for Future Research.

A substantial amount of follow-on work remains to be done in abstract interface specification of VLSI designs, let alone in the application of software technology to VLSI design change management, for which several broad research projects were enumerated in chapter 1. Here are some of the tasks needed to carry on the work in abstract interface specification.

5.2.1 Evaluate Approach's Usefulness with Other Base Languages.

The abstract interface specification method described in this dissertation used CSP as a medium of expression, preferring CSP to standard programming languages because of its capabilities for segregating scheduling concerns from functional. Other modern languages for concurrent programming have been introduced: would the use of these languages, rather than CSP, make representation of abstract interface specifications significantly easier? In particular, the VHSIC Hardware Description Language (VHDL) [Dewe84, Shah85] uses Ada[2] control structures both for multitasking and for modular decomposition. At the same time, the VHDL is a more hardware-oriented language than CSP, suggesting that it might be possible to use the proposed technique for specification and yet specify more compactly with richer primitives.

5.2.2 Investigate Approach's Usefulness for Top-Down Design.

Circuit designers, as section 2.3 notes, are most comfortable with clear-box specifications, a specification technique that leads to bottom-up design. It would be revealing to discover how easily designs could be constructed top-down in practice using the proposed approach: for example, by assigning a desirable capacitance and clock rate to a module pin before the module's internals had been laid out and simulated to learn these parameters.

5.2.3 Specify Global Constraints Better.

At present, there is no efficient way of specifying constraints global to the design module (e.g., power consumption, current draw, tessellation, aspect ratio) using the proposed approach. I believe this difficulty to be minor relative to the gains in practical scalability, complexity control, and change management which the proposed approach achieves; nevertheless, the perceived cost-effectiveness of the approach would be enhanced by a solution.

---

[2] Ada is a trademark of the U.S. Department of Defense

5.2.4 Exploit Adherence Concepts.

Quantizing the utility of specification adherence, as suggested herein, is a concept that could be exploited further. To do this, adherence functions need to be better understood and to be easier to use. One possible way to begin this would be to develop by experience a general set of adherence functions and rules for their interpretation and use. Based on the existence of such a set of functions, meaning could be assigned to k-adherence for particular values of k, so that, for example, "90%-adherence" would have a commonly-understood meaning in the design community. Other related research topics include investigating the effect of composition on k-adherence and determining whether adherence functions are the best way of expressing the value of adherence.

5.2.5 Study Use of Approach for Wider Range of Designs.

It would be useful to learn the applicability of the proposed approach in retroactively specifying interfaces of designs originally implemented in technologies other than nMOS and in design styles other than synchronous, full-custom. The usefulness of the method for specifying interfaces of designs intended for analog implementation should also be investigated.

5.2.6 Refine Approach with Feedback from a Good Implementation.

I fully agree with Brooks [Broo82] in his statement that the most productive software engineering principle to be developed in the last ten years is the "incremental build" approach of Harlan Mills [Myer84]. With this approach, rather than to conduct additional theoretical research, I believe it would now be far more productive to refine the ideas expressed in this dissertation using feedback from the VLSI design community. To do this, one should develop and field a set of good tools implementing the proposed approach, tools that include a well-engineered designer interface. Without such a set of tools and designer interface, it is fruitless to expect that already-overextended designers can be induced to use the proposed approach sufficiently to render a fair evaluation.

One should not, however, underestimate the cost of such tool development, a cost several times the cost of constructing a mere "snapped-together" implementation of the specification principles proposed herein [Broo75]. I began this research proposing to build and collect feedback from such a set of tools, and it was not until I had invested numerous weeks in them that I

realized they would, without an order of magnitude more effort, be unacceptably clumsy and "unmarketable" (even *gratis*) to VLSI designers. Despite the current interest in reducing such costs through capital-intensive methods [Wegn84], the costs of building prototype design tools that are effective in obtaining essential feedback remain a major current impediment to methodological research. This statement leads to my final suggestion.

5.2.7 Further Evaluate Approach's Extensibility.

It is common for Ph.D. dissertations to include in their final section the phrase, "The extensibility of the proposed approach to very large designs at an acceptable cost has not yet been demonstrated." This dissertation is no exception; such demonstration of extensibility, although in a final sense "the proof of the pudding," is well beyond dissertation scope. Meta-research is needed, now that complexity and matters of scale are dominant concerns in so many engineering research fields, to determine ways of verifying, at acceptable cost, the extensibility of proposed techniques to real-world-size problems.

Nevertheless, in the absence of methods for proving extensibility, it would be wrong to give up. There is much precedent for the subsequent and successful large-scale application of ideas that germinated, without certainty of extension, on a modest scale in a laboratory. It is too early to tell whether the ideas set forth herein will enjoy such application, but I am thankful to have had the opportunity to develop them.

APPENDIX A

SPECIFICATIONS OF MODULES IN TEST SET

Contents[1]

Figure	Module	Semantic Level	Syntactic Level
B-6,7	plashin	functional	immaterial
3-6	plashin	four-strength	immaterial
A-1	plashin	drive-load	immaterial
A-2	plashin	four-strength	pingrid
A-3	fnblk	drive-load	immaterial
A-4	malu	drive-load	immaterial
A-5	wasp	functional	immaterial

[1] See Table 4-1 for module descriptions, Table 3-2 for level descriptions. (The first two specifications appear in Appendix B and in Chapter 3, respectively.)

plashin
Module Specification

ASSUMPTIONS

Synopsis: LSSD memory element: static memory cell that can convert to a shift register.
Size: delta x = 18 lambda; delta y = 90 lambda
Description-01: Data is shifted from left to right.
Description-02: p2sh = p2 AND sh; p2shbar = p2 AND (NOT sh); p1shbar = p1 OR (NOT sh)
Notes-01: Control signals can be generated by cell rwcntl.
Related_Documentation: See manual pages for rwcntl and LSSD memory.
Tesselation: Cells abut vertically to form registers of arbitrary length that shift from left to right.
Origin-01: Originally derived from Stanford Cell Library cell PlaShiftIn (CIF ID 81).
Origin-02: Minor modifications by Hedlund (UNC) and Lospinuso (UNC).

SPECIFICATION LEVEL

Semantic: drvload
Syntactic: immaterial

PINS

Name	AccFn	C/D
scin	G_ScanIn	C
scout	P_ScanOut	D
routbar	P_DataOut	D
p2sh	S_Shift	D
p2shbar	S_Hold	D
p1shbar	S_Recirculate	D

INTERNAL STATE VARIABLES

left
right

STATE VARIABLE ATTRIBUTE INITIALIZATIONS

SCIN.maxloadcurr = 275
SCOUT.mindrivecurr = 2750
ROUTBAR.mindrivecurr = 2750
P2SH.maxloadcurr = 300
P2SHBAR.maxloadcurr = 300
P1SHBAR.maxloadcurr = 350
left.mindrivecurr = 2750
left.maxloadcurr = 275
right.mindrivecurr = 2750
right.maxloadcurr = 275

Figure A-1. *plashin* Specification.

```
ACCESS FUNCTIONS

G_ScanIn: none

P_DataOut:
#/*************************************************************/
#/*      Access function P_DataOut:
# */
! routbar = ROUTBAR → skip;

P_ScanOut:
#/*************************************************************/
#/*      Access function P_ScanOut:
# */
! scout = SCOUT → skip;

S_Shift:
#/*************************************************************/
#/*      Access function S_Shift:
# */
? P2SH = p2sh →
# SS = signal(P2SH);
# ST = signal_threshold(P2SH);
 | SS >= ST →
        ? SCIN = scin;
#       steer(SCIN,P2SH,left);
#       inv(left,ROUTBAR);
 | SS < ST → skip;
 |

S_Hold:
#/*************************************************************/
#/*      Access function S_Hold:
# */
? P2SHBAR = p2shbar →
# SH = signal(P2SHBAR);
# ST = signal_threshold(P2SHBAR);
 | SH >= ST →
#       steer (SCOUT,P2SHBAR,left);
 | SH < ST → skip;
 |

S_Recirculate:
#/*************************************************************/
#/*      Access function S_Recirculate:
# */
? P1SHBAR = p1shbar →
# SR = signal(P1SHBAR);
# ST = signal_threshold(P1SHBAR);
 | SR >= ST →
#       steer (ROUTBAR,P1SHBAR,right);
#       inv (right, SCOUT);
 | SR < ST → skip;
 |
```

Figure A-1. *plashin* Specification (Continued).

```
DATA TYPE DEFINITIONS
/* Data Type Definition:
 * Drive-Load semantic refinement level.
 *  This is a very simple electrical model with
 *  symmetric rise/fall times.
 * Immaterial syntactic refinement level.
 *
 * At this level the state variable data structure is
 *      typedef struct stvar {
 *         int sigvartyp;
 *         int sigvoltage;
 *         int sigcurrent;
 *         int mindrivecurr;/* Semantic attribute */
 *         int maxloadcurr; /* Semantic attribute */
 *         } STVAR ;
 * with
 *      sigvartyp = variable type (= drive-load/immaterial);
 *      sigvoltage = drive/load voltage (millivolts)
 *      sigcurrent = average drive/load switching current
 *            (nanoamps)
 *      mindrivecurr = specified minimum drive current
 *            (nanoamps)
 *      maxloadcurr = specified maximum load current
 *            threshold (nanoamps)
 */

#define VHIMAX 5000
#define VHIMIN 3800
#define VTH 3750
#define VLOMAX 1200
#define VLOMIN 250

abs(i)
        int i;
{
return ((i > 0) ? i : -i);
}

inv(stinput,stoutput)
   STVAR *input, *output;
{
output→sigvartyp = input→sigvartyp;
if ((input→sigvoltage <= VTH)
   && (-input→sigcurrent >= output→maxloadcurr)) {
        output→sigvoltage = VHIMAX;
        output→sigcurrent = output→mindrivecurr;
        }
else if ((input→sigvoltage >= VHIMIN)
   && (input→sigcurrent >= output→maxloadcurr)) {
        output→sigvoltage = VLOMIN;
        output→sigcurrent = -output→mindrivecurr;
        }
else {
   output→sigvoltage = undefined (VLOMIN, VHIMAX);
   output→sigcurrent =
```

Figure A-1. *plashin* Specification (Continued).

```
        undefined (-output→mindrivecurr,
           output→mindrivecurr);
   }
return;
}

signal(input)
   STVAR *input;
{
return(abs(input→sigcurrent));
}

signal_threshold(input)
   STVAR *input;
{
return(input→maxloadcurr);
}

steer(input,control,output)
   STVAR *input, *control, *output;
{
output→sigvartyp = input→sigvartyp;
if ((control→sigvoltage >= VHIMIN)
   && (control→sigcurrent >= control→maxloadcurr)) {
        output→sigvoltage = input→sigvoltage - VTH;
        output→sigcurrent = input→sigcurrent;
        }
else {
   output→sigvoltage = undefined (VLOMIN, VHIMAX);
   output→sigcurrent =
         undefined (-control→mindrivecurr,
            control→mindrivecurr);
return;
}

undefined (lo, hi)
        int lo, hi;
/* 'undefined' represents a condition specified as
 * illegal. It returns a value for the purposes of
 * executability in the specification interpretation.
 * This value should either cause the interpretation
 * to terminate or it should cause aberrant behavior,
 * in either case alerting the designer to potential
 * difficulty.
 */
return ((hi-lo)*0.5);
```

Figure A-1. *plashin* Specification (Continued).

ADHERENCE FUNCTION DEFINITIONS

Access Fn f *Adherence Function α[f]*

P_DataOut $\alpha(q,x) =$

$$= \min_{i=1}^{n} \frac{x \rightarrow sig_i}{x0 \rightarrow sig_i},$$

where

$sig_1, sig_2, \ldots, sig_n$

are the n variables in the *sig* data type,
$x0 = P\_DataOut(q)$, and
$0 \leq x \leq x0$;

$= 1$, otherwise.

P_ScanOut $\alpha(q,x) =$

$$= \min_{i=1}^{n} \frac{x \rightarrow sig_i}{x0 \rightarrow sig_i},$$

where

$sig_1, sig_2, \ldots, sig_n$

are the n variables in the *sig* data type,
$x0 = P\_ScanOut(q)$, and
$0 \leq x \leq x0$;

$= 1$, otherwise.

S_Shift

Let

$$S\_Shift\ (q,s) = \left\{ v_1, v_2, \ldots, v_k \right\}$$

and

$$S\_Shift\ (q',s) = \left\{ v'_1, v'_2, \ldots, v'_k \right\},$$

where $\{v_i\}$ is the set of values, of the module state variables, that make up the module state q. Then

$$\alpha(q,s,q') = \max\left(0, \min\left(1, \min_{j=1}^{k} \min_{i} \frac{v_j \rightarrow sig_i}{v_j \rightarrow sig_i}\right)\right).$$

Figure A-1. *plashin* Specification (Continued).

S_Hold

Let

$$S\_Hold\,(q,h) = \left\{ v_1,\ v_2,\ \ldots,\ v_k \right\}$$

and

$$S\_Hold\,(q',h) = \left\{ v'_1,\ v'_2,\ \ldots,\ v'_k \right\},$$

where $\{v_i\}$ is the set of values, of the module state variables, that make up the module state q. Then

$$\alpha(q,h,q') = \max\,(0,\ \min\,(1,\ \min_{j=1}^{k}\ \min_{i}\ \frac{v_j \rightarrow sig_i}{v'_j \rightarrow sig_i}\ )\ ).$$

S_Recirculate

Let

$$S\_Recirculate\,(q,r) = \left\{ v_1,\ v_2,\ \ldots,\ v_k \right\}$$

and

$$S\_Recirculate\,(q',r) = \left\{ v'_1,\ v'_2,\ \ldots,\ v'_k \right\},$$

where $\{v_i\}$ is the set of values, of the module state variables, that make up the module state q. Then

$$\alpha(q,r,q') = \max\,(0,\ \min\,(1,\ \min_{j=1}^{k}\ \min_{i}\ \frac{v_j \rightarrow sig_i}{v'_j \rightarrow sig_i}\ )\ ).$$

Note: The adherence functions defined above are the Linear Ramp (LR) functions introduced in section 3.4. Because their definitions are (unfortunately) complex, we shall abbreviate them as follows in the remainder of this appendix:

α[f] defined for	will be abbreviated as
P_DataOut	LRI [P_DataOut] (q,x)
S_Shift	LRD [S_Shift] (q,s,q')

Figure A-1. *plashin* Specification (Continued).

plashin
Module Specification

ASSUMPTIONS

(Same as for previous specification (A-1))

SPECIFICATION LEVEL

Semantic: Four-strength
Syntactic: Pingrid

PINS

(Same as for previous specification (A-1))

INTERNAL STATE VARIABLES

(Same as for previous specification (A-1))

STATE VARIABLE ATTRIBUTE INITIALIZATIONS

SCIN.varxpos = 0	SCIN.varypos = 2
SCOUT.varxpos = 6	SCOUT.varypos = 2
ROUTBAR.varxpos = 1	ROUTBAR.varypos = 21
P2SH.varxpos = 0	P2SH.varypos = 0
P2SHBAR.varxpos = 0	P2SHBAR.varypos = 1
P1SHBAR.varxpos = 0	P1SHBAR.varypos = 18

ACCESS FUNCTIONS

(Same as for previous specification (A-1))

DATA TYPE DEFINITIONS

```
/* Data Type Definition:
 * Four-strength semantic refinement level.
 * Pingrid syntactic refinement level.
 *
 * At these levels the state variable data structure is
 *       typedef stvar {
 *          int sigvartyp;
 *          int sigfval;
 *          int sigstrength;
 *          int varxpos; /* Syntactic Attribute   */
 *          int varypos; /* Syntactic Attribute   */
 *          } STVAR ;
 * with
 *       sigvartyp = variable type (= four-strength/pingrid);
 *       sigfval = functional value
 *         in {0,1,9 (= undefined)};
 *       sigstrength = signal strength in
 *             {0 (= unknown),
 *                1 (= floating),
 *                2 (= steered),
 *                3 (= restored)}
 * with signal strength codes semantically ordered
 *       (i.e. i > j → strength i > strength j);
 *       varxpos = x state variable grid position;
```

Figure A-2. *plashin* Specification.

```
*       varypos = y state variable grid position.
*/
```

(Operator definitions same as for Figure 3-6)

ADHERENCE FUNCTION DEFINITIONS

(Semantic adherence functions identical to those given for Figure 3-6. Additional functions follow for measuring syntactic adherence. Note that the evaluation of these functions is independent of module state, and that therefore this evaluation can take place statically.)

Access Fn f	Adherence Function $\alpha[f]$	
P_DataOut	$\alpha(q,s)$	= 1, s→varxpos = ROUTBAR→varxpos and s→varypos $\leq$ ROUTBAR→varypos + 1; = 0 otherwise.
P_ScanOut	$\alpha(q,s)$	= 1, s→varypos = SCOUT→varypos and s→varxpos $\leq$ SCOUT→varxpos + 1; = 0 otherwise.
S_Shift	$\alpha(q,s)$	= 1, s→varypos = P2SH→varypos and s→varxpos $\geq$ P2SH→varxpos - 1; = 0 otherwise.
S_Hold	$\alpha(q,s)$	= 1, s→varypos = P2SHBAR→varypos and s→varxpos $\geq$ P2SHBAR→varxpos - 1; = 0 otherwise.
S_Recirculate	$\alpha(q,s)$	= 1, s→varypos = P1SHBAR→varypos and s→varxpos $\geq$ P1SHBAR→varxpos - 1; = 0 otherwise.

Figure A-2. *plashin* Specification (Continued).

fnblk
Module Specification

ASSUMPTIONS

Synopsis: ALU function block

Size: delta x = 42 lambda; delta y = 80 lambda

Description: (M. Lospinuso) The function block receives two inputs, the data register (BUFOUT) and the accumulator (ACOUT). The logic operation performed by the function block is specified by the four function block control lines f0, f1, f2, and f3. The function block operations are listed below:

controls f0 1 2 3	output	mnemonic	controls f0 1 2 3	output	mnemonic
0 0 0 0	0	zero	1 0 0 0	(B+A)′	BnorA
0 0 0 1	BA′	BandA′	1 0 0 1	A′	A′
0 0 1 0	BA	BandA	1 0 1 0	(BxorA)′	BxnorA
0 0 1 1	B	B	1 0 1 1	(B′ A)′	BorA′
0 1 0 0	B′ A	B′ andA	1 1 0 0	B′	B′
0 1 0 1	BxorA	BxorA	1 1 0 1	(BA)′	BnandA
0 1 1 0	A	A	1 1 1 0	(BA′)′	B′ orA
0 1 1 1	B+A	BorA	1 1 1 1	1	one

Related_Documentation: WAfer-Scale Systolic Processor (WASP) Project Technical Note, "dp2 Data Path Module Specifications," July 5, 1984

Origin-01: WASP 1.0, K. Hedlund (UNC)

SPECIFICATION LEVEL

Semantic: Drive-load
Syntactic: Immaterial

PINS

Name	AccFn	C/D	
acout	S_acout	D	
acoutbar	S_acoutbar	D	
bufout	S_bufout	D	
bufoutbar	S_bufoutbar	D	
fnout	P_fnout	D	
sbin	P_sbin	D	
f[B]	S_f[B]	D	B=0..3

INTERNAL STATE VARIABLES

t[B], B=0..3

Figure A-3. *fnblk* Specification.

```
STATE VARIABLE ATTRIBUTE INITIALIZATIONS
    ACOUT.maxloadcurr = 275
    ACOUTBAR.maxloadcurr = 275
    BUFOUT.maxloadcurr = 275
    BUFOUTBAR.maxloadcurr = 275
    FNOUT.mindrivecurr = 5000
    SBIN.mindrivecurr = 50000
    F[B].maxloadcurr = 275, B=0..3
    t[B].mindrivecurr = 2750, B=0..3
    t[B].maxloadcurr = 275, B=0..3

ACCESS FUNCTIONS

P_fnout:
#/*************************************************************/
#/*     Access function P_fnout:                              */
! fnout = FNOUT → skip;

P_sbin:
#/*************************************************************/
#/*     Access function P_sbin:                               */
! fnout = FNOUT → skip;

S_acout:
#/*************************************************************/
#/*     Access function S_acout:                              */
? ACOUT = acout →
#  SA = signal(ACOUT);
#  ST = signal_threshold(ACOUT);
  [ SA >= ST →
#       steer(F1,ACOUT,t);
#       steer(t,BUFOUTBAR,t1);
#       steer(F2,BUFOUT,t);
#       steer(t,ACOUT,t2);
#       arbitrate4 (t0,t1,t2,t3,FNOUT);
#       samenode (FNOUT,SBIN);
  | SA < ST → skip;
  |

S_acoutbar:
#/*************************************************************/
#/*     Access function S_acoutbar:                           */
? ACOUTBAR = acoutbar →
#  SA = signal(ACOUTBAR);
#  ST = signal_threshold(ACOUTBAR);
  [ SA >= ST →
#       steer(F0,BUFOUTBAR,t);
#       steer(t,ACOUTBAR,t0);
#       steer(F3,BUFOUT,t);
#       steer(t,ACOUTBAR,t3);
#       arbitrate4 (t0,t1,t2,t3,FNOUT);
#       samenode (FNOUT,SBIN);
  | SA < ST → skip;
  |
```

Figure A-3. *fnblk* Specification (Continued).

```
S_bufout:
#/*************************************************************/
#/*      Access function S_bufout:                            */
? BUFOUT = bufout →
#  SA = signal(BUFOUT);
#  ST = signal_threshold(BUFOUT);
   [ SA >= ST →
#        steer(F2,BUFOUT,t);
#        steer(t,ACOUT,t2);
#        steer(F3,BUFOUT,t);
#        steer(t,ACOUTBAR,t3);
#        arbitrate4 (t0,t1,t2,t3,FNOUT);
#        samenode (FNOUT,SBIN);
   | SA < ST → skip;
   ]

S_bufoutbar:
#/*************************************************************/
#/*      Access function S_bufoutbar:                         */
? BUFOUTBAR = bufoutbar →
#  SA = signal(BUFOUTBAR);
#  ST = signal_threshold(BUFOUTBAR);
   [ SA >= ST →
#        steer(F0,BUFOUTBAR,t);
#        steer(t,ACOUTBAR,t0);
#        steer(F1,ACOUT,t);
#        steer(t,BUFOUTBAR,t1);
#        arbitrate4 (t0,t1,t2,t3,FNOUT);
#        samenode (FNOUT,SBIN);
   | SA < ST → skip;
   ]

S_f0:
#/*************************************************************/
#/*      Access function S_f0:                                */
? F0 = f0 → skip;

S_f1:
#/*************************************************************/
#/*      Access function S_f1:                                */
? F1 = f1 → skip;

S_f2:
#/*************************************************************/
#/*      Access function S_f2:                                */
? F2 = f2 → skip;
```

Figure A-3. *fnblk* Specification (Continued).

```
S_f3:
#/*************************************************************/
#/*     Access function S_f3:                                 */
? F3 == f3 → skip;
```

DATA TYPE DEFINITIONS

(Same as for Figure A-1, with the following operations added:)

```
arbitrate4 (var1,var2,var3,var4,varout)
   STVAR *var1,*var2,*var3,*var4,*varout;
{
/* This is a very simple arbitration strategy.              */
varout→sigvoltage =
   max (var1→sigvoltage, var2→sigvoltage,
        var3→sigvoltage, var4→sigvoltage);
varout→sigcurrent =
   max (var1→sigcurrent, var2→sigcurrent,
        var3→sigcurrent, var4→sigcurrent);
varout→sigvartyp = var1→sigvartyp;
}

samenode (var1,var2)
   STVAR *var1, *var2;
{
var2→sigvartyp = var1→sigvartyp;
var2→sigvoltage = var1→sigvoltage;
var2→sigcurrent = var1→sigcurrent;
}
```

ADHERENCE FUNCTION DEFINITIONS

Access Fn f	Adherence Function $\alpha[f]$
S_acout	LRD [S_acout] (q,s,q′)
S_acoutbar	LRD [S_acoutbar] (q,s,q′)
S_bufout	LRD [S_bufout] (q,s,q′)
S_bufoutbar	LRD [S_bufoutbar] (q,s,q′)
S_f0	LRD [S_f0] (q,s,q′)
S_f1	LRD [S_f1] (q,s,q′)
S_f2	LRD [S_f2] (q,s,q′)
S_f3	LRD [S_f3] (q,s,q′)
P_fnout	LRI [P_fnout] (q,s)
P_sbin	LRI [P_sbin] (q,s)

Figure A-3. *fnblk* Specification (Continued).

malu
Module Specification

ASSUMPTIONS

Synopsis: WASP 1.3 Minimal ALU. Performs 16 Boolean functions on two bits, one from the accumulator and one bit from outside the data path. The Minimal ALU is the processing element of the WAfer-Scale Systolic Processor (WASP).

Size: delta x = 229 lambda; delta y = 80 lambda

Description: Major component parts: input buffer (dynamic storage during execute, pseudo-static with *phi2* refresh during instruction load), accumulator (pseudo-static storage), inverter pair, function block, and superbuffer (listed sequentially, left to right).

The function block receives two inputs, the data register (BUFOUT) and the accumulator (ACOUT). The logic operation performed by the function block is specified by the four function block control lines f0, f1, f2, and f3.

The accumulator is a pseudo-static storage cell. Its input can come from the data register *(acextp2)*, the current accumulator *(acrecp2)*, or the function block *(acfnp2)*. In order to keep a known value in the accumulator, one of its three input enable lines must be enabled during every *phi2* phase of an execute cycle. The accumulator value becomes undefined if more than one accumulator control line is simultaneously high during any *phi2* phase of an execute cycle. The accumulator can be cleared with the accumulator clear line *(acclrp2)*. During instruction load the control line *StHldHA2* goes high during each *phi2* phase, and causes refresh of the current accumulator value.

The output superbuffer receives the function block output. (Synopsis and Description written by M. Lospinuso.)

Related_Documentation: WAfer-Scale Systolic Processor (WASP) Project Technical Note, "dp2 Data Path Module Specifications," July 5, 1984

Origin-01: K. Hedlund, M. Lospinuso, and E. Vook (UNC).

SPECIFICATION LEVEL

Semantic: Drive-Load
Syntactic: Immaterial

Figure A-4. *malu* Specification.

PINS

Name	AccFn	C/D
f0	S_f0	D
f1	S_f1	D
f2	S_f2	D
f3	S_f3	D
phi1	S_phi1	D
phi2	S_phi2	D
StHldHA2	S_StHldHA2	D
acfnp2	S_acfnp2	D
acbufp2	S_acbufp2	D
acrecp2	S_acrecp2	D
ClrAcpx	S_ClrAcpx	D
DtOut	P_DtOut	D
DtIn	G_DtIn	C

INTERNAL STATE VARIABLES

ACIN0
ACIN1
BUFIN0
BUFIN1
FNOUT
t[B], B=0..3

STATE VARIABLE ATTRIBUTE INITIALIZATIONS

ACIN0.mindrivecurr = 2750
ACIN0.maxloadcurr = 275
ACIN1.mindrivecurr = 2750
ACIN1.maxloadcurr = 275
BUFIN0.mindrivecurr = 2750
BUFIN0.maxloadcurr = 275
BUFIN1.mindrivecurr = 2750
BUFIN1.maxloadcurr = 275
FNOUT.mindrivecurr = 5000
FNOUT.maxloadcurr = 275
F[B].maxloadcurr = 275, B=0..3
t[B].mindrivecurr = 2750, B=0..3
t[B].maxloadcurr = 275, B=0..3
PHI1.maxloadcurr = 200
PHI2.maxloadcurr = 200
STHLDHA2.maxloadcurr = 275
ACFNP2.maxloadcurr = 275
ACBUFP2.maxloadcurr = 275
ACRECP2.maxloadcurr = 275
CLRACPX.maxloadcurr = 300
SBIN.mindrivecurr = 50000
DTIN.maxloadcurr = 300

Figure A-4. *malu* Specification (Continued).

```
ACCESS FUNCTIONS

G_DtIn: none.

P_DtOut:
#/*************************************************************/
#/*      Access function P_DtOut:                             */
! DtOut = SBIN → skip;

S_f0:
#/*************************************************************/
#/*      Access function S_f0:                                */
? F0 = f0 → skip;

S_f1:
#/*************************************************************/
#/*      Access function S_f1:                                */
? F1 = f1 → skip;

S_f2:
#/*************************************************************/
#/*      Access function S_f2:                                */
? F2 = f2 → skip;

S_f3:
#/*************************************************************/
#/*      Access function S_f3:                                */
? F3 = f3 → skip;

S_StHldHA2:
#/*************************************************************/
#/*      Access function S_StHldHA2:                          */
? STHLDHA2 = StHldHA2 →
# SA = signal(STHLDHA2);
# ST = signal_threshold(STHLDHA2);
  [ SA >= ST →
#   steer(ACIN1,STHLDHA2,ACIN0);
#   steer(BUFIN1,STHLDHA2,BUFIN0);
  | SA < ST → skip;
  ]

S_phi1:
#/*************************************************************/
#/*      Access function S_phi1:                              */
? PHI1 = phi1 →
# SA = signal(PHI1);
# ST = signal_threshold(PHI1);
  [ SA >= ST →
#   steer(ACIN0,STHLDHA2,ACIN1);
#   steer(BUFIN0,STHLDHA2,BUFIN1);
#                 /* Evaluate function block              */
#   inv(ACIN1,ACOUTBAR);
#   inv(BUFIN1,BUFOUTBAR);
#   steer(f0,BUFOUTBAR,t);
#   steer(t,ACOUTBAR,t0);
```

Figure A-4. *malu* Specification (Continued).

```
#   steer(f1,ACIN1,t);
#   steer(t,BUFOUTBAR,t1);
#   steer(F2,BUFIN1,t);
#   steer(t,ACIN1,t2);
#   steer(F3,BUFIN1,t);
#   steer(t,ACOUTBAR,t3);
#   arbitrate4 (t0,t1,t2,t3,FNOUT);
#   samenode (FNOUT,SBIN);
  | SA < ST → skip;
  |

S_phi2:
#/*************************************************************/
#/*      Access function S_phi2:                              */
? PHI2 = phi2 →
# SA = signal(PHI2);
# ST = signal_threshold(PHI2);
  | SA >= ST →
    ? DTIN = DtIn;
#   steer(DTIN,PHI2,BUFIN0);
  | SA < ST → skip;
  |

S_acfnp2:
#/*************************************************************/
#/*      Access function S_acfnp2:                            */
? ACFNP2 = acfnp2 →
# SA = signal(ACFNP2);
# ST = signal_threshold(ACFNP2);
  | SA >= ST →
#   steer(FNOUT,ACFNP2,ACIN0);
  | SA < ST → skip;
  |

S_acbufp2:
#/*************************************************************/
#/*      Access function S_acbufp2:                           */
? ACBUFP2 = acbufp2 →
# SA = signal(ACBUFP2);
# ST = signal_threshold(ACBUFP2);
  | SA >= ST →
#   steer(BUFIN1,ACBUFP2,ACIN0);
  | SA < ST → skip;
  |

S_acrecp2:
#/*************************************************************/
#/*      Access function S_acrecp2:                           */
? ACRECP2 = acrecp2 →
# SA = signal(ACRECP2);
# ST = signal_threshold(ACRECP2);
  | SA >= ST →
#   steer(ACIN1,ACRECP2,ACIN0);
  | SA < ST → skip;
  |
```

Figure A-4. *malu* Specification (Continued).

```
S_ClrAcpx:
#/*****************************************************************/
#/*     Access function S_ClrAcpx:                                */
? CLRACPX = ClrAcpx →
# SA = signal(CLRACPX);
# ST = signal_threshold(CLRACPX);
  [ SA >= ST →
#   steer(GND,CLRACPX,ACIN0);
  | SA < ST → skip;
  |
```

DATA TYPE DEFINITIONS

(Same as for Figure A-3)

ADHERENCE FUNCTION DEFINITIONS

Access Fn f	Adherence Fn f
S_f0	LRD [S_f0] (q,s,q′)
S_f1	LRD [S_f1] (q,s,q′)
S_f2	LRD [S_f2] (q,s,q′)
S_f3	LRD [S_f3] (q,s,q′)
S_phi1	LRD [S_phi1] (q,s,q′)
S_phi2	LRD [S_phi2] (q,s,q′)
S_StHldHA2	LRD [S_StHldHA2] (q,s,q′)
S_acfnp2	LRD [S_acfnp2] (q,s,q′)
S_acbufp2	LRD [S_acbufp2] (q,s,q′)
S_acrecp2	LRD [S_acrecp2] (q,s,q′)
S_ClrAcpx	LRD [S_ClrAcpx] (q,s,q′)
P_DtOut	LRI [P_DtOut] (q,s)

Figure A-4. *malu* Specification (Continued).

wasp
Module Specification

ASSUMPTIONS

Synopsis: WASP 1.3, WAfer-scale Systolic Processor.
Designed by Kye Hedlund, William Hargrove, Margaret Lospinuso, and Eric Vook, University of North Carolina at Chapel Hill, 1984.

For certain special-purpose computations, systolic architectures have the potential for very high performance. A wafer-scale implementation, in which processing elements are connected by a mesh of programmable switches on a single wafer, offers the best means for optimizing speed and reliability in systolic arrays. The smaller size and lower capacitance of on-chip wiring permits higher speed than could be obtained with discrete chips and external wiring. On-chip interconnection also brings greater reliability through a reduction in the number of components, and the programmable switch matrix lessens the impact of faults by permitting the inclusion of redundant processors and local exclusion of bad processors.

WASP 1.3, a prototype systolic wafer-scale processor, has an array of simple processors embedded in a switch matrix. The processors are capable of performing all 16 Boolean functions on two bits. The switches are programmable and can receive and send data in four directions. The processing elements can also be programmed to receive and send data in four different directions. Switch programming allows bad processors to be isolated from the network, while level-sensitive scan design (LSSD) allows switch and processing element microcode to be shifted in at low pin cost.

Statistics:
9 processing elements
40 programmable switches
3860 transistors
2965 nodes
3000 x 3000 lambda
866 cm metal wire
621 cm polysilicon wire
407 cm diffusion wire

Description:
All elements (switches and PE's) of the 7 by 7 array in WASP 1.3 have the following SHARED PROPERTIES:
1. Elements communicate only with their 4 NEAREST NEIGHBORS: (north, east, south, west)
2. Communication between elements is by a 1-BIT SERIAL data stream.
3. Each element contains 8 state bits which indicate its ROUTING, i.e. its selection of input and output streams (see 4, 5). These bits are scanned in when the machine is in LOAD mode (pin name mgl = LOAD) and the shift control for the row containing the element is ACTIVE (pin name shgl<R> = 1, R is row no.). Otherwise the current routing setting of the element is retained.
4. As INPUT, each element reads from a SUBSET of the four directions. The results are not defined when more than one direction is selected at a time.
5. As OUTPUT, each element sends data to a SUBSET of the four directions.

Figure A-5. *wasp* Specification.

6. The selection of inputs and outputs are INDEPENDENT. It is not possible to scan in a new setting for one without also scanning in a new setting for the other.
7. The element does not behave differently if its neighbor in any direction is a switch, pe, or input-output pad pair. (An input-output pad pair always listens to its input and always sends to its output. These pads are numbered d<r1c1>_<r2c2> and effect the sending of a signal from the element at row r1, col c1 to the element at row r2, col c2, where either r1, c1, r2, or c2 is an off-grid index (0 or 8)).

Overview of WASP 1.3 LOGICAL LAYOUT of elements:

1. The rows are numbered 1 to 7 from top (north) to bottom (south). The columns are numbered from 1 to 7 from left (west) to right (east).
2. Arrangement of elements: there are 7 rows of 7 elements each.
3. Pattern of elements:
   Odd rows contain only switches.
   Odd columns contain only switches.
   PE's are at the intersection of even rows with even columns.
4. Due to a limitation in the number of available pads, input-output pad pairs are found only at the ends of even rows or columns.

(Assumptions section written by Margaret F. Lospinuso and Eric R. Vook)

SPECIFICATION LEVEL

Semantic: Functional
Syntactic: Immaterial

PINS

Name	AccFn	C/D	
acclr	S_acclr	D	
mgl	S_mgl	D	
phi1gl	S_phi1gl	D	
phi2gl	S_phi2gl	D	
shglHL1[R]	G_shglHL1[R]	C	R=1..7
acscin[C]	G_acscin[C]	C	C=1..7
acscout[C]	P_acscout[C]	D	C=1..7
d02_12	G_d02_12	C	
d12_02	P_d12_02	D	
d04_14	G_d04_14	C	
d14_04	P_d14_04	D	
d06_16	G_d06_16	C	
d16_06	P_d16_06	D	
d28_27	G_d28_27	C	
d27_28	P_d27_28	D	
d48_47	G_d48_47	C	
d47_48	P_d47_48	D	
d68_67	G_d68_67	C	
d67_68	P_d67_68	D	
d82_72	G_d82_72	C	
d72_82	P_d72_82	D	
d84_74	G_d84_74	C	
d74_84	P_d74_84	D	
d86_76	G_d86_76	C	
d76_86	P_d76_86	D	
d20_21	G_d20_21	C	
d21_20	P_d21_20	D	

Figure A-5. *wasp* Specification.

d40_41	G_d40_41	C
d41_40	P_d41_40	D
d60_61	G_d60_61	C
d61_60	P_d61_60	D

STATE VARIABLES

INDIR[R][C]	R=1..7, C=1..7
OUTDIR[R][C]	R=1..7, C=1..7
FUN[R][C]	R=2,4,6, C=2,4,6
ACBUF[R][C]	R=2,4,6, C=2,4,6
ACREC[R][C]	R=2,4,6, C=2,4,6
ACFN[R][C]	R=2,4,6, C=2,4,6
ACIN0[R][C]	R=2,4,6, C=2,4,6
ACIN1[R][C]	R=2,4,6, C=2,4,6
BUFIN0[R][C]	R=2,4,6, C=2,4,6
BUFIN1[R][C]	R=2,4,6, C=2,4,6
OUTRAIL[R][C]	R=0,2,4,6,8, C=0,2,4,6,8

STATE VARIABLE ATTRIBUTE INITIALIZATIONS

None.

ACCESS FUNCTIONS

```
#/*************************************************************/
#/*     Access function P_acscout[1]:                         */
! acscout(1) = ACSCOUT[1] → skip;
#/*************************************************************/
#/*     Access function P_acscout[2]:                         */
! acscout(2) = ACSCOUT[2] → skip;
#/*************************************************************/
#/*     Access function P_acscout[3]:                         */
! acscout(3) = ACSCOUT[3] → skip;
#/*************************************************************/
#/*     Access function P_acscout[4]:                         */
! acscout(4) = ACSCOUT[4] → skip;
#/*************************************************************/
#/*     Access function P_acscout[5]:                         */
! acscout(5) = ACSCOUT[5] → skip;
#/*************************************************************/
#/*     Access function P_acscout[6]:                         */
! acscout(6) = ACSCOUT[6] → skip;
#/*************************************************************/
#/*     Access function P_acscout[7]:                         */
! acscout(7) = ACSCOUT[7] → skip;
#/*************************************************************/
#/*     Access function S_acclr:                  */
? ACCLR = acclr → skip;
#/*************************************************************/
#/*     Access function S_mgl:                                */
? MGL = mgl → skip;
#/*************************************************************/
#/*     Access function S_phi1gl:                             */
? PHI1GL = phi1gl →
  [ PHI1GL == 1 →
    [ MGL == 1 →
```

Figure A-5. *wasp* Specification (Continued).

```
#       printf ("Execute phase, phi1 entered");
#       for (R=2; R<8; R+=2) { for (C=2; C<8; C+=2) {
#                       /* Compute Output Value                    */
#         steer(ACIN0[R][C],PHI1GL,ACIN1[R][C]);
#         steer(BUFIN0[R][C],PHI1GL,BUFIN1[R][C]);
#         fnblk(FUN[R][C],BUFIN1[R][C],ACIN1[R][C],
#            OUTRAIL[R][C]);
#       }}
#                       /* Send Output, if appropriate  */
#         if ((OUTDIR[1][2]&1) != 0) {
#           get_inswitch(1,2,inswitch);
#           D12_02 = OUTRAIL[inswitch[0]][inswitch[1]];
#           }
#         if ((OUTDIR[1][4]&1) != 0) {
#           get_inswitch(1,4,inswitch);
#           D14_04 = OUTRAIL[inswitch[0]][inswitch[1]];
#           }
#         if ((OUTDIR[1][6]&1) != 0) {
#           get_inswitch(1,6,inswitch);
#           D16_06 = OUTRAIL[inswitch[0]][inswitch[1]];
#           }
#         if ((OUTDIR[2][1]&8) != 0) {
#           get_inswitch(2,1,inswitch);
#           D21_20 = OUTRAIL[inswitch[0]][inswitch[1]];
#           }
#         if ((OUTDIR[4][1]&8) != 0) {
#           get_inswitch(4,1,inswitch);
#           D41_40 = OUTRAIL[inswitch[0]][inswitch[1]];
#           }
#         if ((OUTDIR[6][1]&8) != 0) {
#           get_inswitch(6,1,inswitch);
#           D61_60 = OUTRAIL[inswitch[0]][inswitch[1]];
#           }
#         if ((OUTDIR[7][2]&4) != 0) {
#           get_inswitch(7,2,inswitch);
#           D72_82 = OUTRAIL[inswitch[0]][inswitch[1]];
#           }
#         if ((OUTDIR[7][4]&4) != 0) {
#           get_inswitch(7,4,inswitch);
#           D74_84 = OUTRAIL[inswitch[0]][inswitch[1]];
#           }
#         if ((OUTDIR[7][6]&4) != 0) {
#           get_inswitch(7,6,inswitch);
#           D76_86 = OUTRAIL[inswitch[0]][inswitch[1]];
#           }
#         if ((OUTDIR[2][7]&2) != 0) {
#           get_inswitch(2,7,inswitch);
#           D27_28 = OUTRAIL[inswitch[0]][inswitch[1]];
#           }
#         if ((OUTDIR[4][7]&2) != 0) {
#           get_inswitch(4,7,inswitch);
#           D47_48 = OUTRAIL[inswitch[0]][inswitch[1]];
#           }
#         if ((OUTDIR[6][7]&2) != 0) {
#           get_inswitch(6,7,inswitch);
```

Figure A-5. *wasp* Specification (Continued).

```
#          D67_68 = OUTRAIL[inswitch[0]][inswitch[1]];
#          }
  | MGL == 0 →
#       printf ("Load phase, phi1 entered");
#       for (R=1; R<8; R++) {
         ? SHGLHL1[R] = shglHL1(R);
#       }
#       for (C=1; C<8; C++) {
         ACSCOUT[C] = ACSCIN[C];
#       }
  | ~((MGL==1)||(MGL==0)) → skip;
#                         /* If MGL not defined, nothing  */
#                         /*   is to happen.              */
  |
  | PHI1GL == 0 → skip;
  |
#/*************************************************************/
#/*    Access function S_phi2gl:                              */
? PHI2GL = phi2gl →
  [ PHI2GL == 1 →
    [ MGL == 1 →
#       printf ("Execute phase, phi2 entered");
#       steer(BUFIN1[R][C],ACBUF[R][C],ACIN0[R][C]);
#       steer(OUTRAIL[R][C],ACFN[R][C],ACIN0[R][C]);
#       steer(ACIN1[R][C],ACREC[R][C],ACIN0[R][C]);
#       steer(GND,ACCLR,ACIN0[R][C]);
#       if (INDIR[1][2] == 1)
                ? OUTRAIL[0][2] = d02_12;
#       if (INDIR[1][4] == 1)
                ? OUTRAIL[0][4] = d04_14;
#       if (INDIR[1][6] == 1)
                ? OUTRAIL[0][6] = d06_16;
#       if (INDIR[2][1] == 8)
                ? OUTRAIL[2][0] = d20_21;
#       if (INDIR[4][1] == 8)
                ? OUTRAIL[4][0] = d40_41;
#       if (INDIR[6][1] == 8)
                ? OUTRAIL[6][0] = d60_61;
#       if (INDIR[7][2] == 2)
                ? OUTRAIL[8][2] = d82_72;
#       if (INDIR[7][4] == 2)
                ? OUTRAIL[8][4] = d84_74;
#       if (INDIR[7][6] == 2)
                ? OUTRAIL[8][6] = d86_76;
#       if (INDIR[2][7] == 4)
                ? OUTRAIL[2][8] = d28_27;
#       if (INDIR[4][7] == 4)
                ? OUTRAIL[4][8] = d48_47;
#       if (INDIR[6][7] == 4)
                ? OUTRAIL[6][8] = d68_67;
#       for (R=2; R<8; R+=2) { for (C=2; C<8; C+=2) {
#                         /* Locate Input Source */
#         if (INDIR[R][C] != 0) {
#           get_inswitch(R,C,inswitch);
#           BUFIN0[R][C] =
```

Figure A-5. *wasp* Specification (Continued).

```
#                OUTRAIL[inswitch[0]][inswitch[1]];
#           }
#         else BUFIN0[R][C] = 9;
#       }}
  | MGL == 0 →
#       printf ("Load phase, phi2 entered");
#               /* Documented StHldHA2 signal is      */
#               /* equivalent to PHI2GL & (MGL)'.     */
#       steer(ACIN1[R][C],PHI2GL,ACIN0[R][C]);
#       steer(BUFIN1[R][C],PHI2GL,BUFIN0[R][C]);
#                       /* 'BIT' is a counter which   */
#                       /* keeps track of which bit   */
#                       /* of the 15-bit control stream */
#                       /* is currently being entered.  */
#       if (BIT==15) BIT=0;
#       for (C=1; C<8; C++) {
          ? ACSCIN[C] = acscin(C);
#         for (R=1; R<8; R++) {
#           if (SHGLHL1[R] == 1) {
#                       /* Bit stream is encoded into   */
#                       /*   integers here as follows:  */
#                       /* 0,1,13,14 → BIT 14           */
#                       /* 2,3,11,12 → BIT 12           */
#                       /* 4,5,6,7   → BIT 7            */
#                     /* Others as shown below:*/
#               if (BIT== 7) FUN[R][C] = ACSCIN[C];
#               if (BIT== 8) ACFN[R][C] = ACSCIN[C];
#               if (BIT== 9) ACREC[R][C] = ACSCIN[C];
#               if (BIT==10) ACBUF[R][C] = ACSCIN[C];
#               if (BIT==12) OUTDIR[R][C] = ACSCIN[C];
#               if (BIT==14) INDIR[R][C] = ACSCIN[C];
#       } } }
#       BIT++;
  | ~((MGL==1)||(MGL==0)) → skip;
  |
 | PHI2GL == 0 → skip;
 |
#/*************************************************************/
#/*      Access function P_d12_02:                            */
! d12_02 = D12_02 → skip;
#/*************************************************************/
#/*      Access function P_d14_04:                            */
! d14_04 = D14_04 → skip;
#/*************************************************************/
#/*      Access function P_d16_06:                            */
! d16_06 = D16_06 → skip;
#/*************************************************************/
#/*      Access function P_d27_28:                            */
! d27_28 = D27_28 → skip;
#/*************************************************************/
#/*      Access function P_d47_48:                            */
! d47_48 = D47_48 → skip;
```

Figure A-5. *wasp* Specification (Continued).

```
#/*************************************************************/
#/*     Access function P_d67_68:                             */
! d67_68 = D67_68 → skip;
#/*************************************************************/
#/*     Access function P_d72_82:                             */
! d72_82 = D72_82 → skip;
#/*************************************************************/
#/*     Access function P_d74_84:                             */
! d74_84 = D74_84 → skip;
#/*************************************************************/
#/*     Access function P_d76_86:                             */
! d76_86 = D76_86 → skip;
#/*************************************************************/
#/*     Access function P_d21_20:                             */
! d21_20 = D21_20 → skip;
#/*************************************************************/
#/*     Access function P_d41_40:                             */
! d41_40 = D41_40 → skip;
#/*************************************************************/
#/*     Access function P_d61_60:                             */
! d61_60 = D61_60 → skip;
]
DATA TYPE DEFINITIONS

/* Data Type Definition:
 * Functional semantic refinement level.
 * Immaterial syntactic refinement level.
 *
 * At this level the state variable data structure is
 *       typedef struct stvar {
 *                int sigfval;
 *                } STVAR ;
 * with
 *  sigfval = functional value in {0,1,9 (=undefined)};
 *
 * For efficiency, operations are defined on integers
 *  instead of with reference to this equivalent
 *  structure.
 */

/* Operator Definitions are the same as for Figure B-7,
   with the following operations added:                */

/* 1. Operations on state variables:                   */

fnblk (funindex,stvar1,stvar2,stvarout)
        int funindex;
        int stvar1, stvar2;
        int *stvarout;
{
if ((stvar1==9) || (stvar2==9))
        {
        *stvarout = 9;
        return;
        }
```

Figure A-5. *wasp* Specification (Continued).

```
switch (funindex) {
 case 0:
  *stvarout = 0;
  return;
 case 1:
  *stvarout = stvar1 & (1-stvar2);
  return;
 case 2:
  *stvarout = stvar1 & stvar2;
  return;
 case 3:
  *stvarout = stvar1;
  return;
 case 4:
  *stvarout = (1-stvar1) & stvar2;
  return;
 case 5:
  *stvarout = stvar1 ^ stvar2;
  return;
 case 6:
  *stvarout = stvar2;
  return;
 case 7:
  *stvarout = stvar1 | stvar2;
  return;
 case 8:
  *stvarout = 1-(stvar1 | stvar2);
  return;
 case 9:
  *stvarout = 1-stvar2;
  return;
 case 10:
  *stvarout = 1-(stvar1 ^ stvar2);
  return;
 case 11:
  *stvarout = 1-((1-stvar1) & stvar2));
  return;
 case 12:
  *stvarout = 1-stvar1;
  return;
 case 13:
  *stvarout = 1-(stvar1 & stvar2);
  return;
 case 14:
  *stvarout = 1-(stvar1 & (1-stvar2));
  return;
 case 15:
  *stvarout = 1;
  return;
 default:
  printf("Bad function value %d",funindex);
/* Such error flags are not part of the specification.
 * They are included merely for assistance in detecting
 * errors during verification.  See section 3.3.1.3 for
 * a discussion of differences between specifications
```

Figure A-5. *wasp* Specification (Continued).

```
 * and simulators.
 */
   *stvarout = 0;
   return;
}

/* 2. Local closed subroutines, included for efficiency
 *    to replace in-line subroutines.
 */

get_inswitch(currrow,currcol,inswitch)
        int currrow, currcol;
        int inswitch[2];
{
/*************************************************************
 * Function name: get_inswitch
 * Parameters:  currrow: (input) row of active PE
 *                      or switch
 *              currcol: (input) column of active PE
 *                      or switch
 *              inswitch: (output) row and column of PE
 *                      or switch which provides input
 *                      to the active PE or switch
 * Assumptions: 7x7 grid of switches/PE's
 * External References:
 *              INDIR (matrix of input switch settings)
 *              OUTDIR (matrix of output switch settings)
 * Author:      R. Gross, March 1985
 *************************************************************/
extern int INDIR[][8], OUTDIR[][8];
int border1 = 0, border2 = 8;
int newrow, newcol;

loop:
switch (INDIR[currrow][currcol])
    {
    case 1:                     /* North        */
        newrow=currrow-1;
        newcol=currcol;
        break;
    case 2:                     /* East         */
        newrow=currrow;
        newcol=currcol+1;
        break;
    case 4:                     /* South        */
        newrow=currrow+1;
        newcol=currcol;
        break;
    case 8:                     /* West         */
        newrow=currrow;
        newcol=currcol-1;
        break;
    default:
        printf ("Bad value %d in INDIR[%d][%d].",
          INDIR[currrow][currcol],currrow,currcol);
```

Figure A-5. *wasp* Specification (Continued).

```
        printf ("A value of 1 has been assumed.");
        newrow=currrow-1;
        newcol=currcol;
    }
                                /* Consistency Test */
if ((newrow != border1) && (newcol != border1) &&
   (newrow != border2) && (newcol != border2) &&
   ((invert (INDIR[currrow][currcol])
      & OUTDIR[newrow][newcol]) == 0))
    {
    printf("Inconsistency:");
    printf ("INDIR[%d][%d]=%d, OUTDIR[%d][%d]=%d",
        currrow,currcol,INDIR[currrow][currcol],
        newrow,newcol,OUTDIR[newrow][newcol]);
    }
inswitch[0] = newrow;
inswitch[1] = newcol;
if ((newrow != border1) && (newrow != border2) &&
   (newcol != border1) && (newcol != border2) &&
   (((newrow & 1) != 0) || ((newcol & 1) != 0)))
   {
   currrow = newrow;
   currcol = newcol;
   goto loop;
   }
   return;
}

int
invert (dir)
        int dir;
/*************************************************************
 * Function name: invert
 * Parameters:  dir: (input) 1, 2, 4, or 8, representing
 *                        N,E,S,W respectively.
 * Returns:     Input  1      2      4      8
 *              Output 4      8      1      2
 * Assumptions: Input is valid
 * External References: None
 * Author:      R. Gross, March 1985
 *************************************************************/
{
switch (dir)
  {
  case 1: return (4);
  case 2: return (8);
  case 4: return (1);
  case 8: return (2);
  default: return (0);
  }
}
```

Figure A-5. *wasp* Specification (Continued).

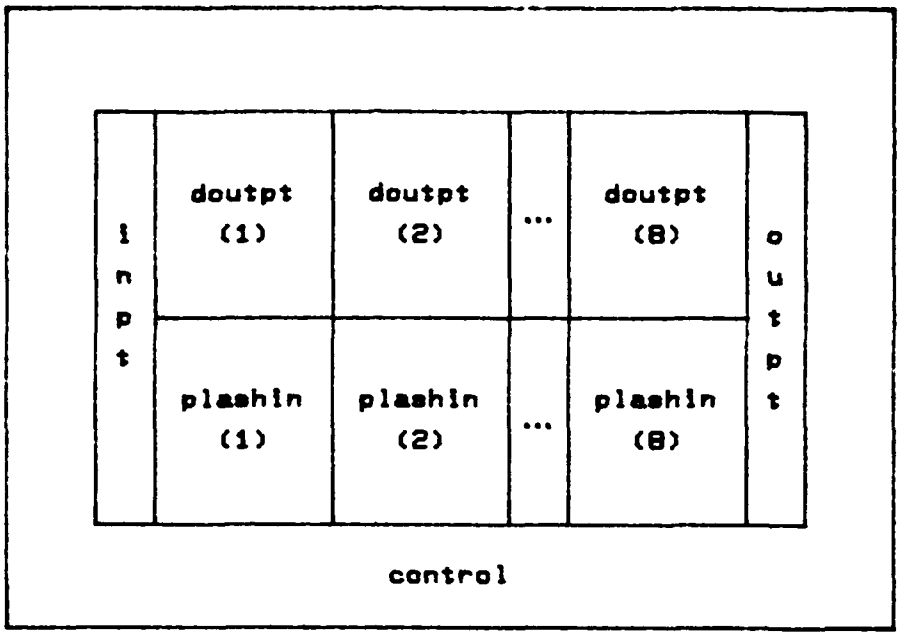
ADHERENCE FUNCTION DEFINITIONS

Access Fn f	Adherence Function $\alpha[f]$	
P_acscout[C] C=1..7	$\alpha(q,s)$	= 1 if s = P_acscout[C]; = 0 otherwise.
S_acclr	$\alpha(q,s,q')$	= 1 if q' = S_acclr(q,s); = 0 otherwise.
S_mgl	$\alpha(q,s,q')$	= 1 if q' = S_mgl(q,s); = 0 otherwise.
S_phi1gl	$\alpha(q,s,q')$	= 1 if q' = S_phi1gl(q,s); = 0 otherwise.
S_phi2gl	$\alpha(q,s,q')$	= 1 if q' = S_phi2gl(q,s); = 0 otherwise.

Figure A-5. *wasp* Specification (Continued).

APPENDIX B

ILLUSTRATION OF SPECIFICATION EXECUTION



To execute this interconnection of eight 'plashin' shift register cells, I used 19 simple module processes. In addition to the 'plashin' module process, which was obtained by copying the specification (see Figure 3-6):

- 'doutpt(i)' reported module-specific output from each register cell;
- 'inpt' and 'outpt' provided and reported register test input and output, respectively; and
- 'control' was an environment process which provided control stimuli.

Each module process was specified independently of the others; only a simple "channel file" *(containing only syntactic information)* was used to describe interconnections. The CSP/84 language, described in section 3.3 of the text, was the medium used for representing both module process specifications and the channel file.

This specification execution was performed at a functional refinement level; with minor modifications, interpretations at a more refined level (not presented here) were also obtained. Also, with suitable type conversion mechanisms included in specification data type definitions, mixed-refinement-level execution was obtained (e.g., 'doutpt' was specified at a lesser refinement level than 'plashin'). The same channel file sufficed throughout.

***

Contents

```
##########################################################
# Name:                 nplashin
#
# Function:     This is a n-plashin-cell interconnection.
#               One must set n in the first statement.
#
# Author:       R. Gross, October 1984
##########################################################
set n    8

array plashin    1:n
array doutpt     1:n

# datapath:
connect inpt.ScOut              to plashin(1).ScIn

for i      2 : n
connect plashin(i-1).ScOut      to plashin(i).ScIn
connect plashin(i-1).DOut       to doutpt(i-1).DIn
endfor

connect plashin(n).ScOut        to outpt.ScIn
connect plashin(n).DOut         to doutpt(n).DIn

# control:
for i      1 : n
connect control.s(i-1)          to plashin(i).s
connect control.r(i-1)          to plashin(i).r
connect control.h(i-1)          to plashin(i).h
connect control.od(i-1)         to doutpt(i).od
endfor
connect control.os              to outpt.os
```

Figure B-1. Channel File.

```
##include <stdio.h>
process control ::
        output port int s(8) ;
        output port int h(8) ;
        output port int r(8) ;
        output port int od(8);
        output port int os;
int CELLS ;
int count ;
char command;
char line[100];
int i;

command = ' ' ;
CELLS = 8 ;

*[ command != 'q' →

# printf ("Enter s for shift, h for hold, q for quit:");
#       fgets(line,100,stdin);
#       command = line[0];

#       /* Shift:                                          */
#       /* stimulate shift (s)                             */
#       /*   and recirculate (r) repeatedly.               */
#       /* o is used to trigger the printed output.        */

        [ command == 's' →
#               for (i = 0; i < CELLS; i++) {
                        ! s(i) = 1;
#                       }
#               for (i = 0; i < CELLS; i++) {
                        ! s(i) = 0;
#                       }
#               for (i = 0; i < CELLS; i++) {
                        ! od(i) = 1;
#                       }
                ! os = 1;
#               for (i = 0; i < CELLS; i++) {
                        ! r(i) = 1;
#                       }
#               for (i = 0; i < CELLS; i++) {
                        ! r(i) = 0;
#                       }
#               for (i = 0; i < CELLS; i++) {
                        ! od(i) = 1;
#                       }
                ! os = 1;
```

Figure B-2. *control* Module Process.

```
#       /* Hold:                                          */
#       /* stimulate hold (h)                             */
#       /*   and recirculate (r) repeatedly.              */
#       /* o is used to trigger the printed output.       */

        | command == 'h' →
#               for (i = 0; i < CELLS; i++) {
                        ! h(i) = 1;
#                       }
#               for (i = 0; i < CELLS; i++) {
                        ! h(i) = 0;
#                       }
#               for (i = 0; i < CELLS; i++) {
                        ! od(i) = 1;
#                       }
                ! os = 1;
#               for (i = 0; i < CELLS; i++) {
                        ! r(i) = 1;
#                       }
#               for (i = 0; i < CELLS; i++) {
                        ! r(i) = 0;
#                       }
#               for (i = 0; i < CELLS; i++) {
                        ! od(i) = 1;
#                       }
                ! os = 1;
        |
|
end process
```

Figure B-2. *control* Module Process (Continued).

```
##include <stdio.h>
process doutpt ::
guarded input port int od;
        input port int DIn ;
int val ;
int O ;

#/* read and print output values                    */

*[ ? O = od →
        [ O == 1 →
                ? val = DIn;
#     printf("Data Output Cell %d value = %d",
#        csp_proc_num, val);
#     fflush (stdout) ;
        [ O == 0 → skip ;
        ]
]
end process
```

Figure B-3. *doutpt* Module Process.

```
##include <stdio.h>
process inpt ::
        output port int ScOut ;

! ScOut = 0;
! ScOut = 0;
! ScOut = 1;
! ScOut = 1;
! ScOut = 0;
! ScOut = 0;
! ScOut = 1;
! ScOut = 1;

*[1 →
        ! ScOut = 0;
]

end process
```

Figure B-4. *inpt* Module Process.

```
##include <stdio.h>
process outpt ::
guarded input port int os;
        input port int ScIn ;
int val;
int O;

#/* read and print output values                    */

*[ ? O = os →
        [ O == 1 →
                ? val = ScIn;
#    printf("Scan Output value = %d", val);
#       fflush (stdout) ;
        [ O == 0 → skip ;
        ]
]
end process
```

Figure B-5. *outpt* Module Process.

```
##include <stdio.h>
process plashin ::
      input port int ScIn ;
guarded input port int s ;
guarded input port int h ;
guarded input port int r ;
guarded output port int ScOut ;
guarded output port int DOut ;

int left ;
int right ;
int DOUT;
int SCIN;
int SCOUT;
int S ;
int H ;
int R ;

#/* Initialize.                                          */
#/*     9 means undefined.                               */
                left = 9;
                right = 9;

#/*     Poll input ports:                                */

*|
#/*****************************************************/
#/*     Access function P_DOut:
# */
! DOut = DOUT → skip;

#/*****************************************************/
#/*     Access function P_ScOut:
# */
! ScOut = SCOUT → skip;

#/*****************************************************/
#/*     Access function S_s:
# */
| ? S = s →
# SS = signal(S);
# ST = signal_threshold(S);
  | SS >= ST →
        ? SCIN = ScIn;
#       steer(SCIN,S,left);
#       inv(left,DOUT);
  | SS < ST → skip;
  |
```

Figure B-6. *plashin* Module Process.

```
#/*************************************************************/
#/*     Access function S_h:
# */
| ? H = h →
# SH = signal(H);
# ST = signal_threshold(H);
  | SH >= ST →
#       steer (SCOUT,H,left);
  | SH < ST → skip;
  ]

#/*************************************************************/
#/*     Access function S_r:
# */
| ? R = r →
# SR = signal(R);
# ST = signal_threshold(R);
  | SR >= ST →
#       steer (DOUT,R,right);
#       inv (right, SCOUT);
  | SR < ST → skip;
  ]
]
end process
```

Figure B-6. *plashin* Module Process (Continued).

```
/* Data Type Definition:
 * Functional semantic refinement level.
 * Immaterial syntactic refinement level.
 *
 * At this level the state variable data structure is
 *      typedef stvar {
 *              int sigvartyp;
 *              int sigfval;
 *              } STVAR ;
 * with
 *      sigvartyp = variable type (= 1)
 *      sigfval = functional value in {0,1,9 (= undefined)}
 */

#define INFINITY 65535

inv(var1,var2)
   STVAR *var1, *var2;
{
var2→sigvartyp = var1→sigvartyp;
var2→sigfval = (var1→sigfval == 9)
              ? 9 : (1 - var1→sigfval);
return;
}

int
signal(var1)
   VAR *var1;
{
return(INFINITY);
}

int
signal_threshold(var1)
   VAR *var1;
{
return(0);
}

steer(var1,var2,var3)
   STVAR *var1, *var2, *var3;
{
var3→sigvartyp = (var2→sigfval == 1)
              ? var1→sigvartyp : var3→sigvartyp ;
var3→sigfval = (var2→sigfval == 1)
              ? var1→sigfval : var3→sigfval ;
return;
}
```

Figure B-7. Data Type Definitions.

(User input preceded by 'UI:')

```
UI: csp -o ptyp.def.o nplashin control
UI: (same line, cont'd) doutpt inpt outpt plashin
    channel file is nplashin
    control:
    doutpt:
    inpt:
    outpt:
    plashin:
    Linking.
    Running.
    Enter s for shift, h for hold, q for quit:
UI: s
    Data Output Cell 1 value = 1
    Data Output Cell 2 value = 9
    Data Output Cell 3 value = 9
    Data Output Cell 4 value = 9
    Data Output Cell 5 value = 9
    Data Output Cell 6 value = 9
    Data Output Cell 7 value = 9
    Data Output Cell 8 value = 9
    Scan Output value = 9
    Data Output Cell 1 value = 1
    Data Output Cell 2 value = 9
    Data Output Cell 3 value = 9
    Data Output Cell 4 value = 9
    Data Output Cell 5 value = 9
    Data Output Cell 6 value = 9
    Data Output Cell 7 value = 9
    Data Output Cell 8 value = 9
    Enter s for shift, h for hold, q for quit:
    Scan Output value = 9
```

Figure B-8. Sample Output.

```
UI: s
    Data Output Cell 1 value = 1
    Data Output Cell 2 value = 1
    Data Output Cell 3 value = 9
    Data Output Cell 4 value = 9
    Data Output Cell 5 value = 9
    Data Output Cell 6 value = 9
    Data Output Cell 7 value = 9
    Data Output Cell 8 value = 9
    Scan Output value = 9
    Data Output Cell 1 value = 1
    Data Output Cell 2 value = 1
    Data Output Cell 3 value = 9
    Data Output Cell 4 value = 9
    Data Output Cell 5 value = 9
    Data Output Cell 6 value = 9
    Data Output Cell 7 value = 9
    Data Output Cell 8 value = 9
    Enter s for shift, h for hold, q for quit:
    Scan Output value = 9
UI: h
    Data Output Cell 1 value = 1
    Data Output Cell 2 value = 1
    Data Output Cell 3 value = 9
    Data Output Cell 4 value = 9
    Data Output Cell 5 value = 9
    Data Output Cell 6 value = 9
    Data Output Cell 7 value = 9
    Data Output Cell 8 value = 9
    Scan Output value = 9
    Data Output Cell 1 value = 1
    Data Output Cell 2 value = 1
    Data Output Cell 3 value = 9
    Data Output Cell 4 value = 9
    Data Output Cell 5 value = 9
    Data Output Cell 6 value = 9
    Data Output Cell 7 value = 9
    Data Output Cell 8 value = 9
    Enter s for shift, h for hold, q for quit:
    Scan Output value = 9
UI: q
    alt fail :line 74,control.c(0). error 1
    write :line 7,inpt.c(0). error 3
```

Figure B-8. Sample Output (Continued).

BIBLIOGRAPHY

[Adri82]. Adrion, W.R., M.A. Branstad, and J.C. Cherniavsky, "Validation, Verification, and Testing of Computer Software." *ACM Computing Surveys 14,* 2 (June 1982), pp. 159-192.

[Albr83]. Albrecht, A.J. and J.E. Gaffney, Jr., "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation." *IEEE Transactions on Software Engineering SE-9,* 6 (November 1983), pp. 639-648.

[Ance83a]. Anceau, F., "CAPRI: A Design Methodology and a Silicon Compiler for VLSI Circuits Specified by Algorithms." In Bryant, R., ed., *Proceedings of the Third Caltech Conference on VLSI,* March 21-23, 1983, pp. 15-31.

[Ance83b]. Anceau, F. and E.J. Aas, eds., *Proceedings, IFIP TC 10/WG 10.5 International Conference on Very Large Scale Integration (VLSI '89).* New York: North-Holland, Elsevier Science Publishers, 1983.

[Bain84]. Bain, I.L., private communication.

[Balz83]. Balzer, R., T.E. Cheatham, Jr., and C. Green, "Software Technology in the 1990's: Using a New Paradigm." *Computer 16,* 11 (November 1983), pp. 39-45.

[Barb82]. Barbacci, M., "An Introduction to ISPS." In [Siew82], pp. 23-32.

[Barr84]. Barrow, H.G., "Proving the Correctness of Digital Hardware Designs." *VLSI Design 5,* 7 (July 1984), pp. 64-77.

[Bart77]. Bartussek, W. and D.L. Parnas, "Using Traces to Write Abstract Specifications for Modules." University of North Carolina at Chapel Hill Computer Science Department Technical Report No. 77-012.

[Bela79]. Belady, L.A. and M.M. Lehman, "The Characteristics of Large Systems." In P. Wegner, ed., *Research Directions in Software Technology.* Cambridge: The MIT Press, 1979, pp. 106-138.

[Bell71]. Bell, C.G. and A. Newell, *Computer Structures: Readings and Examples.* New York: McGraw-Hill, 1971.

[Berg81]. Bergland, G.D., "A Guided Tour of Program Design Methodologies." *Computer 14,* 10 (October 1981), pp. 13-37.

[Blaa76]. Blaauw, G.A., *Digital System Implementation.* Englewood Cliffs, NJ: Prentice-Hall, 1976.

[Blaa81]. Blaauw, G.A. and F.P. Brooks, Jr., *Computer Architecture.* Pre-publication draft, 1981.

[Blaa83]. Blaauw, G.A., private communication, 1983.

[Boeh81]. Boehm, B.W., *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, 1981.

[Boeh83]. Boehm, B.W. and T.A. Standish, "Software Technology in the 1990's: Using an Evolutionary Paradigm." *Computer 16*, 11 (November 1983), pp. 30-37.

[Boeh84]. Boehm, B.W., T.E. Gray, and T. Seewaldt, "Prototyping Versus Specifying: A Multiproject Experiment." *IEEE Transactions on Software Engineering SE-10*, 3 (May 1984), pp. 290-303.

[Brit81a]. Britton, K.H., R.A. Parker, and D.L. Parnas, "A Procedure for Designing Abstract Interfaces for Device Interface Modules." *Proceedings of the 5th International Conference on Software Engineering*, March 9-12, 1981, pp. 195-204. (For Britton's earlier work, see entries under Heninger, K.L.)

[Brit81b]. Britton, K.H. and D.L. Parnas, "A-7E Software Module Guide." *Naval Research Laboratory Memorandum Report 4702*, December 8, 1981.

[Brit83]. Britton, K.H., private communication, 1983.

[Broo75]. Brooks, F.P., Jr., *The Mythical Man-Month*. Reading, MA: Addison-Wesley, 1975.

[Broo82]. Brooks, F.P., Jr., Class notes for COMP 145, "Software Engineering Laboratory," University of North Carolina at Chapel Hill, Spring 1982.

[Cane83]. Canepa, M., E. Weber, and H. Talley, "VLSI in FOCUS: Designing a 32-bit CPU Chip." *VLSI Design 4*, 1 (January-February 1983), pp. 20-24.

[Card81]. Cardelli, L. and G. Plotkin, "An Algebraic Approach to VLSI Design." In Gray, J.P., ed., *Proceedings, International Conference on Very Large Scale Integration (VLSI 81)*. New York: Academic Press, 1981, pp. 173-182.

[Chen83]. Chen, M.C. and C.A. Mead, "A Hierarchical Simulator Based On Formal Semantics." In R. Bryant, ed., *Proceedings of the Third Caltech Conference on VLSI*, March 21-23, 1983, pp. 207-223.

[Chmu82]. Chmura, L.J. and D.M. Weiss, "The A-7E Software Requirements Document: Three Years of Change Data." *Naval Research Laboratory Memorandum Report 4985*, November 8, 1982.

[Clem82]. Clements, P.C., "Interface Specifications for the A-7E Shared Services Module." *Naval Research Laboratory Memorandum Report 4868*, September 8, 1982.

[Clem83]. Clements, P.C., S.R. Faulk, and D.L. Parnas, "Interface Specifications for the SCR (A-7E) Application Data Types Module." *Naval Research Laboratory Report 8734*, August 23, 1983.

[Clem84]. Clements, P.C. *et al.*, "A Standard Organization for Specifying Abstract Interfaces." *Naval Research Laboratory Report 8815*, June 14, 1984.

[Cohe83]. Cohen, B., "On the Impact of Formal Methods in the VLSI Community." In [Ance83b], pp. 469-479.

[Cole85]. Cole, B.C., "Wafer-Scale Faces Pessimism." *Electronics Week 58,* 13 (April 1, 1985), pp. 49-53.

[Curtis]. Curtis, D., "IDS, An Interface Description System." Alcoa, unpublished. (Cited in [Park81].)

[Dall83]. Dallen, J., "The Synthesis and Validation of Experimental VLSI Design Using Decoupled Behavioral, Structural, and Physical Specifications." Ph.D. Dissertation, Duke University Department of Computer Science, 1983.

[Dasg84]. Dasgupta, S., *The Design and Description of Computer Architectures.* New York: John Wiley & Sons, 1984.

[Dewe84]. Dewey, A., "The VHSIC Hardware Description Language." *VLSI Design 5,* 11 (November 1984), pp. 33-39.

[Diet74]. Dietmeyer, D.L., "Introducing DDL." *Computer 7,* 12 (December 1974), pp. 34-38.

[Duss84]. Dussault, J., C-C. Liaw, and M.M. Tong, "Automating the Front End of the Chip Design Process." *VLSI Design 5,* 9 (September 1984), pp. 62-71.

[Evan85]. Evanczuk, S., "Getting Out of the Gate: High-Level Modeling in Circuit Design." *VLSI Design 6,* 1 (January 1985), pp. 60-66.

[Gajs83]. Gajski, D.D. and R.H. Kuhn, "New VLSI Tools." *Computer 16,* 12 (December 1983), pp. 11-14.

[Gray82]. Gray, J.P., I. Buchanan, and P.S. Robertson, "Designing Gate Arrays Using a Silicon Compiler." *ACM IEEE 19th Design Automation Conference Proceedings,* (1982), pp. 377-383.

[Gros83a]. Gross, R.R., "Silicon Compilers: A Critical Survey." In Microelectronics Center of North Carolina Technical Report 83-06, Research Triangle Park, NC, July 1983.

[Gros83b]. Gross, R.R., "Hierarchical Structure for Families of VLSI Designs." University of North Carolina at Chapel Hill Department of Computer Science Working Paper, October 6, 1983.

[Gros83c]. Gross, R.R., "Transfer of Software Methodology to VLSI Design: Technology Status and Outlook." University of North Carolina at Chapel Hill Department of Computer Science Technical Report 83-008, December 1983.

[Gros84]. Gross, R.R., "A Proposed Information-Theoretic Approach to VLSI Design Change Measurement." University of North Carolina at Chapel Hill Department of Computer Science Working Paper, May 28, 1984.

[Hafe83]. Hafer, L.J. and A.C. Parker, "A Formal Method for the Specification, Analysis, and Design of Register-Transfer Level Digital Logic." *IEEE Transactions on Computer-Aided Design CAD-2,* 1 (January 1983), pp. 4-18.

[Hals77]. Halstead, M.V., *Elements of Software Science.* New York: American Elsevier, 1977.

[Heat83]. Heath, L.S., "Criteria for Modular Decomposition in VLSI Layout." University of North Carolina at Chapel Hill Computer Science Department Working Paper, June, 1983.

[Hedl83]. Hedlund, K.S., "WASP--WAfer-scale Systolic Processor." *VLSI Design 4,* 4 (July-August 1983), pp. 70-71.

[Hedl84a]. Hedlund, K.S., "WASP 1.0 Specifications." University of North Carolina at Chapel Hill Department of Computer Science Microelectronics Design Laboratory Working Paper, July 15, 1984.

[Hedl84b]. Hedlund, K.S. and L. Snyder, "Systolic Architectures -- A Wafer Scale Approach." *Proceedings of the 1984 International Conference on Computer Design* (ICCD '84), October 1984, Port Chester, NY, pp. 604-610.

[Heni78]. Heninger, K.L. *et al.,* "Software Requirements for the A-7E Aircraft." *Naval Research Laboratory Memorandum Report 3876,* November 27, 1978.

[Heni80]. Heninger, K.L., "Specifying Software Requirements for Complex Systems: New Techniques and their Application." *IEEE Transactions on Software Engineering SE-6,* 1 (January 1980), pp. 2-13. (For Heninger's later work, see entries under Britton, K.H.)

[Hest81]. Hester, S.D., D.L. Parnas, and D.F. Utter, "Using Documentation as a Software Design Medium." *Bell System Technical Journal 60,* 8 (October 1981), pp. 1941-1977.

[Hoar78]. Hoare, C.A.R., "Communicating Sequential Processes." *Communications of the ACM 21,* 8 (August 1978), pp. 666-677.

[IBM82]. IBM Corporation, Federal Systems Division, "The Design of Complex Systems: Models and Methods." Course Notebook, July 1982.

[Jaza80a]. Jazayeri, M. *et al.,* "Design and Implementation of a Language for Communicating Sequential Processes." *Proceedings, 1980 IEEE International Conference on Parallel Processing (August).*

[Jaza80b]. Jazayeri, M. *et al.,* "CSP/80: A Language for Communicating Sequential Processes." *Proceedings, IEEE COMPCON Fall 1980,* pp. 736-740.

[John82]. Johnson, D.S., "The NP-Completeness Column: An Ongoing Guide." *Journal of Algorithms 3,* pp. 381-395 (1982).

[Karp84]. Karplus, K., "A Formal Model for MOS Clocking Disciplines." Cornell University Department of Computer Science Technical Report TR 84-632, August 1984.

[Katz83]. Katz, R.H. and S. Weiss, "Chip Assemblers: Concepts and Capabilities." *ACM IEEE 20th Design Automation Conference Proceedings,* June 27-29, 1983, pp. 25-30.

[Kish82]. Kishimoto, Z. *et al.,* "The Intersection of VLSI and Software Engineering for Testing and Verification." Workshop Report, VLSI and Software Engineering Workshop, 4-6 October 1982, Port Chester, NY, pp. 10-49.

[Kuni84]. Kunii, T.L., ed., *VLSI Engineering.* Berlin: Springer-Verlag, 1984.

[Latt81]. Lattin, W.W. *et al.,* "A Methodology for VLSI Chip Design." *Lambda 2,* 2 (Second Quarter 1981), pp. 34-44.

[Lehm84]. Lehman, M.M., V. Stenning, and W.M. Turski, "Another Look at Software Design Methodology." *ACM SIGSOFT Software Engineering Notes 9,* 2 (April 1984), pp. 38-53.

[Leve82]. Levene, A.A. and G.P. Mullery, "An Investigation of Requirement Specification Languages: Theory and Practice." *Computer 15,* 5 (May 1982), pp. 50-59.

[Lieb85]. Lieberherr. K.J., "Toward a Standard Hardware Description Language." *IEEE Design and Test of Computers 2,* 1 (February 1985), pp. 55-62.

[Lien83]. Lientz, B.P., "Issues in Software Maintenance." *ACM Computing Surveys 15,* 3 (September 1983), pp. 271-278.

[Lipp83]. Lipp, H.M., "Methodical Aspects of Logic Synthesis." *Proceedings of the IEEE 71,* 1 (January 1983), pp. 88-97.

[Lipt83]. Lipton, R.J. *et al.,* "VLSI Layout as Programming." *ACM Transactions on Programming Languages and Systems 5,* 3 (July 1983), pp. 405-421.

[Lisk75]. Liskov, B.H. and S.N. Zilles, "Specification Techniques for Data Abstractions." *IEEE Transactions on Software Engineering 1,* 1 (March 1975), pp. 7-19.

[Lisk79]. Liskov, B.H. and V. Berzins, "An Appraisal of Program Specifications." In P. Wegner, ed., *Research Directions in Software Technology.* Cambridge, MA: The MIT Press, 1979.

[Mari78]. Marino, E., "Computer Interface Description." *Proceedings of the 17th Annual Technical Symposium,* National Bureau of Standards and ACM, June 1978.

[Maru85]. Maruyama, F. and M. Fujita, "Hardware Verification." *Computer 18,* 2 (February 1985), pp. 22-32.

[McCa76]. McCabe, T.J., "A Complexity Measure." *IEEE Transactions on Software Engineering SE-2,* 4 (December 1976), pp. 308-320.

[Mead80]. Mead, C.A., and L. Conway, *Introduction to VLSI Systems.* Reading, MA: Addison-Wesley, 1980.

[Mead83]. Mead, C.A., "Structural and Behavioral Composition of VLSI." In [Ance83b], pp. 3-8.

[Meye85]. Meyer, B., "On Formalism in Specifications." *IEEE Software 2,* 1 (January 1985), pp. 6-26.

[Midd84]. Middleton, D., private communication, 1984.

[Midd85]. Middleton, D., "CSP/85 Manual." University of North Carolina at Chapel Hill Department of Computer Science Technical Report TR85-010, June 1985.

[Moln85]. Molnar, C.E., T-P. Fang, and F.U. Rosenberger, "Synthesis of Delay-Insensitive Modules." In H. Fuchs, ed., *1985 Chapel Hill Conference on Very Large Scale Integration,* May 15-17, 1985, pp. 67-86.

[Mona82]. Monachino, M., "Design Verification System for Large-Scale LSI Designs." *IBM Systems Journal 26,* 1 (January 1982), pp. 89-99.

[Moor84]. Moore, G.E., "A Macro View of Microelectronics." *IEEE Design and Test of Computers 1,* 4 (November 1984), pp. 15-23.

[Mudg81]. Mudge, J.C., "VLSI Chip Design at the Crossroads." In Gray, J.P., ed., *Proceedings, International Conference on Very Large Scale Integration (VLSI 81).* New York: Academic Press, 1981, pp. 205-215.

[Murp83]. Murphy, B.T., "Microcomputers: Trends, Technologies, and Design Strategies." *IEEE Journal of Solid-State Circuits SC-18,* 3 (June 1983), pp. 236-244.

[Musa85]. Musa, J.D., "Software Engineering: The Future of a Profession." *IEEE Software 2,* 1 (January 1985), pp. 55-62.

[Myer84]. Myers, W., "Can Software Development Processes Improve — Drastically?" *IEEE Software 1,* 3 (July 1984), pp. 101-102.

[Nash84]. Nash, J.D., "Bibliography of Hardware Description Languages." *ACM SIGDA Newsletter 14,* 1 (February 1984), pp. 18-37.

[NPS79]. Naval Postgraduate School, Monterey, CA, *Software Engineering Principles.* (Course Notes, 11-22 June 1979.)

[Newk83]. Newkirk, J. and R. Mathews, *The VLSI Designer's Library.* Reading, MA: Addison-Wesley, 1983.

[Noic82]. Noice, D., R. Mathews, and J. Newkirk, "A Clocking Discipline for Two-Phase Digital Systems." Proceedings, *IEEE International Conference on Circuits and Computers,* September 1982, pp. 108-111.

[Park81]. Parker, A.C. and J.J. Wallace, "SLIDE: An I/O Hardware Description Language." *IEEE Transactions on Computers C-30,* 6 (June 1981), pp. 423-439.

[Parn72a]. Parnas, D.L., "A Technique for Software Module Specification with Examples." *Communications of the ACM 15,* 5 (May 1972), pp. 330-336.

[Parn72b]. Parnas, D.L., "On the Criteria to be Used in Decomposing Systems into Modules." *Communications of the ACM 15,* 12 (December 1972), pp. 1053-1058.

[Parn74]. Parnas, D.L., "On a 'Buzzword': Hierarchical Structure." *Proceedings of the IFIP Congress 1974,* pp. 336-339.

[Parn75a]. Parnas, D.L. and G. Handzel, "More on Specification Techniques for Software Modules." Technical Report 75/1, Research Group on Operating Systems I, Technische Hochschule, Darmstadt, W. Germany, February 13, 1975.

[Parn75b]. Parnas, D.L., "Software Engineering or Methods for Multi-Person Construction of Multi-Version Programs." In Goos, G. and J. Hartmanis, eds., *Programming Methodology,* Lecture Notes in Computer Science 23, Berlin: Springer-Verlag, 1975.

[Parn76a]. Parnas, D.L., "On the Design and Development of Program Families." *IEEE Transactions on Software Engineering SE-2,* 1 (March 1976), pp. 1-9.

[Parn76b]. Parnas, D.L. and H. Wuerges, "Response to Undesired Events in Software Systems." *Proceedings of the 2nd International Conference on Software Engineering,* October 13-15, 1976, pp. 437-446.

[Parn77a]. Parnas, D.L., "The Use of Precise Specifications in the Development of Software." In B. Gilchrist, ed., *Information Processing 1977* (IFIP Congress Proceedings), pp. 861-867.

[Parn77b]. Parnas, D.L., "Use of Abstract Interfaces in the Development of Software for Embedded Computer Systems." *Naval Research Laboratory Memorandum Report 8047,* June 3, 1977.

[Parn83a]. Parnas, D.L., *et al.,* "Interface Specifications for the SCR (A-7E) Extended Computer Module." *Naval Research Laboratory Memorandum Report 4848,* January 6, 1983.

[Parn83b]. Parnas, D.L., P.C. Clements, and D.M. Weiss, "Enhancing Reusability With Information Hiding." *Proceedings of the Workshop on Reusability in Programming,* September 1983, pp. 240-247.

[Parn84]. Parnas, D.L., "The Modular Structure of Complex Systems." *Proceedings of the 7th International Conference on Software Engineering,* Orlando, FL, March 1984, pp. 408-417. Reprinted in *IEEE Transactions on Software Engineering SE-11,* 3 (March 1985), pp. 259-266.

[Part83]. Partsch, H. and R. Steinbrüggen, "Program Transformation Systems." *ACM Computing Surveys 15,* 3 (September 1983), pp. 199-236.

[Patt84]. Patterson, D.A., "VLSI Systems Building: A Berkeley Perspective." *1984 Conference on Advanced Research in VLSI, M.I.T.,* pp. 84-91.

[Penf72]. Penfield, P., Jr., "Description of Electrical Networks Using Wiring Operators." *Proceedings of the IEEE 60,* 1 (January 1972), pp. 49-53.

[Pilo85]. Piloty, R. and D. Borrione, "The Conlan Project: Concepts, Implementations, and Applications." *Computer 18,* 2 (February 1985), pp. 81-92.

[Poul84]. Poulton, J., "PXPL 4.0 Pin Typing Conventions." University of North Carolina at Chapel Hill Department of Computer Science Microelectronics Systems Laboratory Working Paper, July 15, 1984.

[Poul85]. Poulton, J., *et al.,* "PIXEL-PLANES: Building a VLSI-Based Graphic System." In H. Fuchs, ed., *1985 Chapel Hill Conference on Very Large Scale Integration,* May 15-17, 1985, pp. 35-60.

[Rade82]. Rader, J., ed., *Proceedings of the Computer Society VLSI and Software Engineering Workshop,* Port Chester, NY, October 4-6, 1982.

[Rem83]. Rem, M., J.L.A. van de Snepscheut, and J.T. Udding, "Trace Theory and the Definition of Hierarchical Components." In R. Bryant, ed., *Proceedings of the Third Caltech Conference on VLSI,* March 21-23, 1983, pp. 225-239.

[Roma85]. Roman, G.-C., "A Taxonomy of Current Issues in Requirements Engineering." *Computer 18,* 4 (April 1985), pp. 14-23.

[Rose84]. Rosenberg, J.B., "Chip Assembly Techniques for Custom IC Design in a Symbolic Virtual-Grid Environment." *1984 Conference on Advanced Research in VLSI, M.I.T.,* pp. 213-225.

[Rows80]. Rowson, J.A., "Understanding Hierarchical Design." Ph.D. Thesis, California Institute of Technology Computer Science Department Report No. 3710, April 1980.

[SST84]. Seattle Silicon Technology, Inc., *Concorde Databook* (Preliminary). Bellevue, WA: 1984.

[Séga81]. Ségal, R., "Structure, Placement, and Modelling." California Institute of Technology Computer Science Department Technical Report No. 4029, February 1, 1981.

[Séqu83]. Séquin, C. H., "Managing VLSI Complexity: An Outlook." *Proceedings of the IEEE 71,* 1 (January 1983), pp. 149-166.

[Shah85]. Shahdad, M. *et al.,* "VHSIC Hardware Description Language." *Computer 18,* 2 (February 1985), pp. 94-103.

[Shan49]. Shannon, C.E. and W. Weaver, *The Mathematical Theory of Communication.* Urbana: University of Illinois Press, 1949.

[Shaw84]. Shaw, M., "Abstraction Techniques in Modern Programming Languages." *IEEE Software 1,* 4 (October 1984), pp. 10-26.

[Shee84]. Sheeran, M., "muFP, a Language for VLSI Design." *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, Austin, Texas (August 5-8, 1984), pp. 104-112.

[Siew82]. Siewerek, D.P., C.G. Bell, and A. Newell, *Computer Structures: Principles and Examples*. New York: McGraw-Hill, 1982.

[Sisk82]. Siskind, J.M., J.R. Southard and K.W. Crouch, "Generating Custom High Performance VLSI Designs from Succinct Algorithmic Descriptions." *MIT Conference on Advanced Research in VLSI*, 1982, pp. 28-40.

[Smit82]. Smith, C.U. and J. Dallen, "Future Directions for VLSI and Software Engineering." Duke University Department of Computer Science Technical Report CS-1982-16, 1982.

[Smit83]. Smith, C.U. and J. Dallen, "A Comparison of Design Strategies for Software and for VLSI." Duke University Department of Computer Science Technical Report CS-1982-25, September 1982. Also in *Proceedings, COMPCON Spring 83, pp. 268-268.*

[Stef82a]. Stefik, M. *et al.*, "The Partitioning of Concerns in Digital System Design." *MIT Conference on Advanced Research in VLSI*, 1982, pp. 43-52.

[Stef82b]. Stefik, M. and L. Conway, "Towards the Principled Engineering of Knowledge." *The AI Magazine 3*, 3 (Summer 1982), pp. 4-16.

[Subr83]. Subrahmanyam, P.A., "Synthesizing VLSI Circuits from Behavioral Specifications: A Very High Level Silicon Compiler and its Theoretical Basis." In [Ance83b], pp. 195-210.

[Suzu85]. Suzuki, N., "Concurrent Prolog as an Efficient VLSI Design Language." *Computer 18*, 2 (February 1985), pp. 33-40.

[Teic77]. Teichroew, D. and E.A. Hershey, III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems." *IEEE Transactions on Software Engineering SE-3*, 1 (January 1977), pp. 41-48. Reprinted in E. Yourdon, ed., *Classics in Software Engineering*. New York: Yourdon Press, 1979.

[Tham84]. Tham, K., R. Willoner and D. Wimp, "Functional Design Verification by Multi-Level Simulation." *ACM IEEE 21st Design Automation Conference Proceedings*, June 25-27, 1984, pp. 473-478.

[Trim81]. Trimberger, S. *et al.*, "A Structured Design Methodology and Associated Software Tools." *IEEE Transactions on Circuits and Systems CAS-28*, 7 (July 1981), pp. 618-634.

[Tsai83]. Tsai, L.L. and J.O. Achugbue, "BURLAP: A Hierarchical VLSI Design System." *VLSI Design 4*, 4 (July-August 1983), pp. 21-26.

[Uddi84]. Udding, J.T., "Classification and Composition of Delay-Insensitive Circuits." Doctoral Dissertation, Eindhoven University of Technology, September 1984.

[Valt82]. Valtorta, M., "The Linear Placement Problem." In "Course Projects on VLSI Algorithmics," Duke University Department of Computer Science Technical Report CS-1982-17, 1982.

[Viss76]. Vissers, C., "Interface, A Dispersed Architecture." *Proceedings of the Third Annual Symposium on Computer Architecture,* 1976, pp. 98-104.

[Wall84]. Wallich, P., "The One-Month Chip: Design." *IEEE Spectrum 21,* 9 (September 1984), pp. 30-34.

[Wegn84]. Wegner, P., "Capital-Intensive Software Technology." *IEEE Software 1,* 3 (July 1984), pp. 7-45.

[Wern83]. Werner, J., "The Moving Target." *VLSI Design 4,* 2 (March/April 1983), p. 8.

[Wern84]. Werner, J., "Halt the Proliferation of HDLs." *VLSI Design 5,* 2 (February 1984), p. 10.

[Wirt71]. Wirth, N., "Program Development by Stepwise Refinement." *Communications of the ACM 14,* 4 (April 1971), pp. 221-227.

[Witt85]. Witt, B.I., "Communicating Modules: A Software Design Model for Concurrent Distributed Systems." *Computer 18,* 1 (January 1985), pp. 67-77.

[Yeh83]. Yeh, R.T., "Software Engineering." In E.A. Torrero, ed., "Tomorrow's Computers," *IEEE Spectrum 20,* 11 (November 1983), pp. 91-94.

[Zade75]. Zadeh, L.A. *et al., Fuzzy Sets and their Applications to Cognitive and Decision Processes.* New York: Academic Press, 1975.

[Zade84]. Zadeh, L.A., "Making Computers Think Like People." *IEEE Spectrum 21,* 8 (August 1984), pp. 26-32.

[Zave84]. Zave, P., "The Operational Versus The Conventional Approach to Software Development." *Communications of the ACM 27,* 2 (February 1984), pp. 104-118.

[Zelk84]. Zelkowitz, M.V. *et al.,* "Software Engineering Practices in the US and Japan." *Computer 17,* 6 (June 1984), pp. 57-66.

END

FILMED

10-85

DTIC