University of North Carolina at Chapel Hill
Department of Computer Science
Technical Report
TR84-007

FINAL TECHNICAL REPORT
SRC Contract 82-11-003
Transfer of Software Methodology to VLSI Design

Frederick P. Brooks, Jr.
with
Major Richard R. Gross, USAF
Lenwood S. Health

September 20, 1984

Final Technical Report
SRC Contract 82-11-003
Transfer of Software Methodology to VLSI Design

Goals.

This investigation was proposed as a high-risk one-year exploration of VLSI design methods by a group of software engineers, to see how applicable the complexity management techniques evolved over fifteen years of software engineering would be to the analogous VLSI design complexity problems. We said we would look at the problem for a year and then report whether there was gold in these hills.

Results - Quick Net

1. There is not enough gold there to interest any of our faculty investigators in personally exploring further; it is less promising than our other research opportunities. So we made no application for further support.

Professor Kye Hedlund is pursuing independent work on VLSI design methodology that has been influenced by our research under this contract, although it had other principal stimuli.

2. There is enough gold that one of our students, Major Richard R. Gross, USAF, has chosen to write his Ph.D. dissertation on the hot topic developed from this contract research. His subject is:

"Using Software Technology to Manage Change in VLSI Design."

His Ph.D. dissertation advisor is Professor Peter Calingaert, co-principal investigator of this contract.

Since Major Gross is fully supported by the Air Force, his work, both during the life of the SRC contract and over the coming year, has cost SRC nothing for salaries. The contract did pay part of the cost of a computer workstation which Major Gross shares.

An invited paper by Major Gross, delivered at the Seventh Syracuse-Maryland Minnowbrook Workshop on Software Performance Evaluation, July, 1984 is attached. Major Gross's dissertation will be delivered when finished, about June, 1985.

3. During the contract, Professor Hedlund, one of the investigators, received a two year IBM Young Faculty Research Award in nation-wide competition.

4. Mr. Lenny Heath, a research assistant on this contract, became interested in problems in the theory of 3-D VLSI, in particular, the problem of how many layers are required to ensure wireability. He has lowered the previous best upper bound from nine "pages" to seven for the "book embedding" problem.

His paper "Embedding Planar Graphs in Seven Pages", has been accepted for the 25th Symposium on the Foundations of Computer Science and will be published in the Proceedings in October. A ten-page extended abstract is attached.

Results - Discussion

Dimensionality. We found software engineering (SE) complexity management techniques to be less applicable than we had expected. I think this arises from two causes. The most important is that the interface problem between two VLSI subdesigns has three (vector) dimensions, whereas software interfaces have one vector dimension, a parameter vector describing a *Logical*, or *information*. interface. The components of the interface vector are data objects, which themselves may have quite complex structure and representation. These components may pass across the interface explicitly, as in a procedure call, or implicitly, as global variables.

VLSI sub-design interfaces have this same Logical dimension, with the same data objects being passed. Except where two sub-designs each also have an interface to a third which has memory, all logical passing must be explicit rather than implicit, but this doesn't seem to be an important restriction.

VLSI interfaces also have a second dimension, the Spatial vector with "2 1/2" subdimensions. The wires coming out of one component have to align, in two subdimensions, and in layer, a half-subdimension, with those going into its neighbor. The alignment points are often called *pins*.

This is not merely a resource allocation restriction, as in software, where components are commonly rationed as to how much space they may occupy. In VLSI, the shape of the space may also be constrained. The Spatial dimension of the interfaces is precisely *geometric*, not merely topological, and requires coordinates.

The third vector dimension of VLSI is the Electrical. For each pin the driving or driven impedance, the voltages representing 0 and 1, and the timing and wave shapes of all pulses must be defined. Maximum current may also be constrained.

The Electrical dimension, like the Spatial, is also subject to over-all resource rationing: total power, total current allowed for a subcomponent.

I believe this essential difference in dimensionality means that VLSI design methodology will take an independent path from software design methodology, except in certain respects illuminated below, most notably management of change.

Flow. We studied a second major difference between the ways VLSI and software designs are assembled from their component sub-designs.

In software, the assembly paradigm is that of the nesting of called procedures. Even where run-time efficiency demands static binding and open subroutines, the logical model is that of nesting, at least within major system components. It is not uncommon for a pipelining paradigm to be used between major components (consider Unix "pipes"), but the "structured" SE methodologies make neither use of nor provision for this paradigm.

In VLSI, on the other hand, wire costs more than logic, up-stream data-paths slow cycle time, and unidirectional, parallel dataflow is most efficient in time and space. Only open subroutines make sense; closed ones cost too much in communications. Major Gross believes this dissimilarity will vanish as software addresses parallel computation. Maybe so, but I think it will endure a long time. Moreover, all of the past SE technology that we hoped to harness reflects it fundamentally to the core.

Single-Language Design. In my view one of the most promising VLSI design ideas I have seen evolved from computer engineering, not software engineering. Professor Gerrit

A. Blaauw of the Twente Technical Institute in the Netherlands presented to us his current work on VLSI design. He uses a single language, a constrained standard APL, to describe first the functional specifications, then the boundary interfaces, and finally the logical design (the *implementation*) of the chip. The same approach extends to electrical and timing specifications. The use of one language allows formal verification that the implementation indeed implements what the architecture specifies. His approach is an extension of that set forth in his book, *Digital System Implementation*, Prentice-Hall, 1976.

If I were continuing research in VLSI design I should certainly pursue this promising approach.

Management of Change. As we noted in our second quarterly report, we found one area of high promise for further work. This is the transfer and adaptation of D.L. Parnas's methodology for the anticipation and management of change in design.

With computer programs, as with all established design disciplines such as those for buildings, bridges, airplanes, and computer hardware, totally new designs are rather rare. Most designs are extensions, reworks, modifications of previous designs. Indeed, even new designs heavily borrow sub-part and component designs from early ones.

Hence the most important design management task is that for this evolution, revision, extension and adaptation process -- i.e., the management of change.

The VLSI discipline is so new that this evolutionary process is not yet evident -- most designers are still iterating on their first designs, and most designs are new. Hence Gross's topic is exceedingly timely just now, as the VLSI design discipline moves into the evolutionary stage.

Moreover, of all the work done in software engineering, I feel that Parnas's work on change management is the most specific, is the most concrete, is the most revolutionary in concept, and offers the most benefit to the VLSI design discipline.

We are pleased that this research approach is developing and look forward to Major Gross's results. We very much appreciate SRC's research support and hope that both our negative results and our positive ones will bear further fruit in the VLSI design community.

cc:
MCNC
SRC
UNC Office of Research Administration
UNC Contracts and Grants

University of North Carolina At Chapel Hill
Department of Computer Science

Technical Report
TR84-005

APPLICATION OF SOFTWARE CHANGE MANAGEMENT TECHNOLOGY TO VLSI DESIGN

Major Richard R. Gross, USAF
September 1984

Invited paper. Presented at the Seventh Syracuse University/University of Maryland Minnowbrook Workshop on Software Performance Evaluation, Blue Mountain Lake, NY, July 24-27, 1984.

This research was sponsored in part by Semiconductor Research Corporation Grant No. 41258. The views expressed in this paper are those of the author and do not reflect the official policy or position of the Department of Defense of the U.S. Government.

APPLICATION OF SOFTWARE CHANGE MANAGEMENT TECHNOLOGY TO VLSI DESIGN

Major Richard R. Gross, USAF
Air Force Institute of Technology and
The University of North Carolina at Chapel Hill
September 1984

Abstract

This paper reports on the results of an eighteen-month investigation conducted to determine potentially fruitful areas of application of software engineering techniques to Very Large Scale Integration (VLSI) design. The management of change is found to be a particularly crucial such area, and it is postulated that the extension to VLSI design of certain software change management techniques developed by D.L. Parnas would have positive results. The extension, implementation, and assessment task for these techniques, however, is substantial, and research subtasks leading to its successful completion are therefore identified and ordered.

Contents:

***

I. Introduction.

A. The Question to be Explored.

In scientific research, a frequent source of breakthroughs is the application of an established answer from one field of endeavor to an unanswered question which exists in a similar field. Similarities between software engineering and VLSI design became apparent soon after the advent of the latter as a discipline in the early 1980s. Several papers and at least one workshop [Rade82] were devoted to enumerating such similarities as the following:

-- Both fields possess a computer heritage; indeed, many VLSI applications consist of the implementation of former software functions in silicon.

– Complexity management is a driving problem in both fields. Complexity increases superlinearly as projects become large.
– In both fields, abstraction is important as a tool in dealing with complexity.
– Both fields increasingly recognize the importance of a life-cycle focus. In particular, the processes of managing and communicating the effects of change are critical to both fields.
– Both fields currently emphasize accelerated design evaluation because errors caught early are less costly to repair.
– As perhaps the most striking similarity, improved tools and methodologies are seen in both fields as a remedy for their outstanding problems.

Is there reason, then, to believe that techniques which have been used successfully to address software engineering problems can be extended or adapted to apply to counterpart problems in VLSI design? If so, what are some of the fruitful areas of application?

B. The Scope and Approach of the Exploration.

To investigate these questions, a research group was formed at the University of North Carolina, Chapel Hill in 1983 to perform an eighteen-month "modest-scale exploration." The group's purpose was to identify areas, if any, which appeared particularly promising for further research into transferring software technology to VLSI design. The research approach employed was as follows:

– Assemble a team of professionals from the software and VLSI design communities.
– Present a typical VLSI design problem to the team as a stimulus for thought on potential applications of software engineering methodology. Postulate theses about such applications.
– Evaluate the theses and assign the most promising to team members for study at greater length.
– Critique the results and suggest future directions for research in this area.

C. The Purpose of this Paper.

This paper is a final report on the exploration described in the preceding paragraph, presenting the group's conclusions and suggestions for future work.

The group's general conclusions were as follows:

1. The similarities between the two fields which were enumerated earlier were confirmed. Further, even some apparently obvious dissimilarities faded upon closer examination:

– In VLSI, the model of efficient computation is pipelining, not nesting. Further, unlike classical software, VLSI is concerned with a unidirectional flowing structure, parallelism, and open subroutines. Nevertheless, lately the software field is also confronting to a much greater degree the issues involved in parallelism. This dissimilarity will probably not endure.

– In VLSI, different cost factors place a premium on earlier testing. There is no counterpart in software to the costly fabrication step in integrated-circuit manufacture. Nevertheless, software is also now emphasizing earlier testing and rapid prototyping. This dissimilarity also is diminishing in significance.

– A fundamental difference seems to be that VLSI design has more concerns than just function, that the VLSI design space is multidimensional, concerned with geometry and electricity as well as function. However, while it has traditionally been held that functional specification is adequate to define software completely, there is now at least some evidence that orthogonal axes (such as quality) also exist in the software design space [Weis84]. It is true, however, that classical software engineering has dealt primarily only with function, so that in general current software engineering principles may have to be extended to deal with the additional design concerns of VLSI.

2. Further technology transfer in both directions between software engineering and VLSI design appears feasible. The most important catalyst for this process will be a re-initiation of the dialogue between what are now largely separate research communities.

3. The following observation of Boehm [Boeh81] applies *a fortiori* to VLSI design:

> The most important software engineering skills we must learn are the skill involved in dealing with a plurality of goals which may be at odds with each other, and the skill of coordinating the application of a plurality of means, each of which provides a varying degree of help or hindrance in achieving a given goal, depending upon the situation.

4. A most crucial problem in both fields is the management of change. There is evidence that at least two software engineering concepts for change management, information hiding and families of designs, can be extended to VLSI design.

5. Techniques for precise specification of abstract VLSI interfaces are needed before the benefits of either information hiding or families of designs can be realized, however.

In the remainder of this paper, the last two of these conclusions, which formed the focus of the group's effort, will be elaborated.

II. Change Management: An Increasingly Critical Problem.

A. Problem Description.

Currently, the stated major concern in VLSI design is the management of complexity [Mudg81, Séqu83]. Traditional techniques for complexity management, such as hierarchy, restriction, and structuring, have been partially effective in the VLSI context; nevertheless, the design process is still excessively 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 [Wern83a], and its seriousness stems from the fact that it is particularly sensitive to the increases in complexity which 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 I am unaware of similar studies directed specifically at VLSI systems evolution, there is good reason to suspect that the effects of progressive changes on such 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, however, has a compounded impact on increasing costs of communication and thus of design, making the cost factors of design in the multi-designer environment significantly different from those which 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.

B. Relevant Work -- D.L. Parnas's Change Management Techniques.

Modern software methodologies, which mostly emphasize design *per se*, offer little promise of assisting VLSI designers in managing change. Few, if any, of them also address the factor of change in the software life-cycle. In this regard the software design techniques of D.L. Parnas are conspicuously exceptional.

1. Information Hiding. Parnas is perhaps best known for introducing the concept of "information hiding" [Parn72], a means of encapsulating design details that are likely to change so that the effect of the change, when it occurs, is limited. He begins by defining an *interface* between two programs [Brit81a] 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." A *secret*, on the other hand, is a set of assumptions about the internal operation of one of the programs that the other programmer is *not* allowed to make. Information hiding, then, is performing the modular decomposition of the programming system in such a way as to encapsulate into separate modules those aspects of the system that are likely to change, i.e., the secrets. The interfaces, by extension, consist of those assumptions which are less likely to change, with the result that the effects of likely changes to the system are compartmented, affecting small numbers of modules only.

Information hiding seems deceptively simple, disguising the fact that making the decisions involved in applying it requires both substantial knowledge of the project and of the underlying technologies. Also, there is likely to be a run-time cost associated with the strict encapsulation of, for example, data structures: every access to the data they contain will necessitate a call to the encapsulating module. Nevertheless, the results of a recent extensive test of information hiding, conducted in the Software Cost Reduction Project [Brit81b, Chmu82] at the Office of Naval Research, indicate that the information hiding principle imparts a discipline to system design that is well worth the costs it engenders.

2. Hierarchical Structuring for Design Families. A second way in which Parnas provides for design change is by examining the concept of design *families*, characterizing [Parn76] 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." Parnas's key assumption is that attention to family similarities during all stages of the design process will lead to designs that are relatively easy to modify (to obtain other family members); thus, if one of the other family members is the desired result of a proposed change, the cost of such a modification will be reduced substantially. Brooks [Broo82] has suggested, for example, that the following are possible variances among family members:

(1) Designs can be functionally identical but make different resource tradeoffs.
(2) Designs can be subsets of the same (super)program or subsets of each other.
(3) Designs can be built on a common base ("kernel") but provide different user interfaces to meet varying needs.
(4) Designs can have similar façades (architectures) and different implementations.

It is tempting to ask, therefore, whether or not considering a set of VLSI designs as such a family has the potential to reduce the cost of VLSI design development and maintenance.

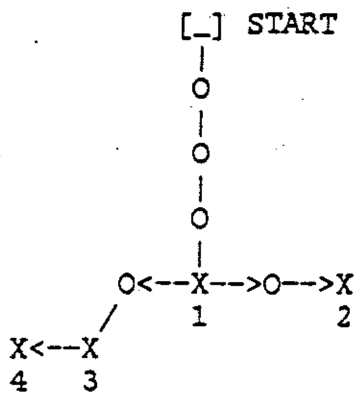

Figure 1. Sequential Completion [Parn76].
(C) 1976 IEEE. Used by permission.

Symbols: [_] is the set of initial possibilities;
O is the incomplete program;
X is the working program.

VLSI design is not now characterized by family considerations. Perhaps the most common approach to VLSI design, in fact, is what Parnas [Parn76] calls "sequential completion." (Figure 1). In sequential completion, a single design is developed and then iteratively reworked as necessary to obtain the desired result. Every iteration of such a process produces a finished design which is an ancestor of future iterations. Consequently, every design shares some characteristics of its ancestors, whether or not they remain appropriate, simply because they are too difficult to "work out" of the design. Designs developed in this way are non-robust under change because of the propagated complexity of the constructs which must be included in the design solely to address the obsolete ancestral features.

Parnas, however, suggests that this sacrifice of robustness is not necessary. Rather than producing new designs from previously completed ones, he advocates the deliberate and stepwise development of "intermediate designs" (or "incomplete designs") which are never intended to be implemented but which instead encapsulate conscious (abstract) design decisions. Such intermediate designs define families (in that each can be a design ancestor of a family of designs which share its characteristics) and thus provide a hierarchy, the *design decision hierarchy,* of checkpoints for design backtracking (Figure 2). If a change is desired, the work done in developing the intermediate design need not be repeated.

Observe that the order in which abstract design decisions are made and encapsulated in intermediate representations defines the design decision hierarchy. The breadth of the families created is determined by the ordering of the decisions made during the design process. Therefore, one would like the earliest design decisions to be those most likely to be invariant, so that the decision hierarchy is developed in order of increasing likelihood of change.

3. Precise Specification. To be useful in change management, both the modular decomposition embodying information hiding and the intermediate designs defining families must be precisely represented. Parnas proposes two means for such representation. The first is stepwise refinement, the well-known embodiment [Wirt71] of Dijkstra's concept of abstract machines. The second is primarily due to Parnas himself, and is called "module specification." In module specification, no pseudo-code is produced: rather, the intermediate design is decomposed into independent program groups called "modules" which are represented by a careful specification of their external (black-box) interfaces. While such decomposition and specification are both costly and demanding, they permit a true separation of concerns (potentially very useful in multi-person design development) and allow the freedom to postpone inter-module connectivity considerations from the early stages of design. (The benefit of this freedom in VLSI design, admittedly, seems dubious.)

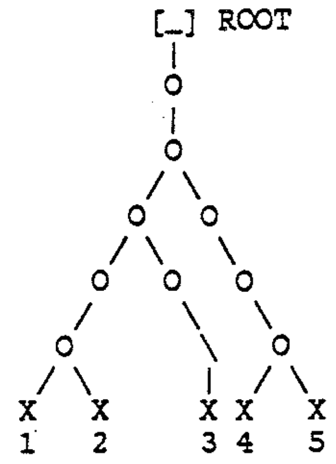

Figure 2. Intermediate Designs [Parn76].
(C) 1976 IEEE. Used by permission.

Symbols: [_] is the set of initial possibilities;
O is the incomplete program;
X is the working program.

In summary, then, the following points should be noted:

- Parnas's concepts of information hiding and design families may contribute to reducing the cost of design development and maintenance.
- Both information hiding and design families *encourage* the postponement of change-prone decisions.
- Precise specification of designs is necessary to obtain the benefits of either technique.

C. Component Research Problems.

How might the promise of the Parnas techniques be further tested? The group postulated that following approach might be used:

- Extension of Parnas's Techniques to VLSI Design. An initial task would be to extend Parnas's techniques of information hiding, hierarchical structuring for families of designs, and precise specification into the VLSI domain. Such an extension has in turn three subtasks:
  - Identifying relevant VLSI design decisions;
  - Ordering (and then making) these decisions to obtain information hiding and broad design families; and
  - Precisely representing/specifying these design decisions.
- Implementation of Parnas's Techniques in a VLSI Design Framework. Once Parnas's techniques were extended into the VLSI domain, new or revised design methods incorporating these techniques would be required before the techniques' utility could be tested by experimentation. Again, three subtasks are involved:
  - Establishing criteria of merit for design decisions;
  - Developing design aids incorporating these decision criteria; and
  - Implementing design techniques which use the evaluated decisions.
- Evaluation of the Utility of Parnas's Techniques in Managing Change. Finally, the techniques must be evaluated through experimentation designed to assess whether or not their application provides measurably effective VLSI design change management.

These tasks define a research program substantial in scope. Before such a program could be completed, background research would need to be conducted in several areas. Some of these areas are defined below:

1. Abstract Interface Specification. Recall that Britton, Parker, and Parnas [Brit81] have defined 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.

Surprisingly, in spite of the greater levels of modularization of IC designs with increasing levels of integration, little has been done in developing techniques for precise characterization and specification of abstract interfaces for VLSI designs. One contributing factor is the greater dimensionality of the VLSI interface, so that more concerns than merely function must be included in the specification.

Of the existing methodologies for hardware interface specifications, the research group was most impressed with the approach taken by Blaauw [Blaa76]. Blaauw's use of the language APL to specify hardware interfaces achieves an unusual economy of discourse, capitalizing on the careful and well-established semantics of APL and on the applicability of APL to both architecture and implementation descriptions.

2. Quantifying Ease of Change. One criticism which has been justly raised about methodological research is that claimed benefits for proposed methods are frequently unsubstantiated. To avoid this criticism, one would like to have a technique-independent 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, McCabe, or Albrecht and Gaffney [Albr83]). Results indicated that techniques employing such approaches will probably need to be quite complex in order to capture design information content successfully. Further, there may well be assumptions embedded in the design that render desirable information components uncountable, suggesting that such metrics must be derived from a design representation which also embodies these assumptions. An abstract interface specification for the design is such a representation.

3. Information Hiding. The decomposition of a VLSI design into modules is an important phase of the design process. In a study of decomposition criteria conducted under the auspices of this exploration, Heath [Heat83] found that information hiding 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." Consequently, precise interface specification techniques for VLSI designs are essential if the benefits of information hiding are to be obtained.

4. Creating Broad Design Families. During the course of this exploration, I studied whether or not considering the successive representations of an evolving VLSI design as a family had the potential to reduce the cost of design development and maintenance [Gros83]. While the conclusion was affirmative, exploiting the family concept depends critically on the availability of precisely-specified intermediate designs which 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.

5. Determining Extent of Required Revalidation. The decision to change a VLSI design brings about the following activities:

– Determine the nature and scope of the change required.
– Perform the change.
– Ensure, by testing, the correctness of the design 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 one of the chief reasons that information hiding was developed in the software domain. An extension of information hiding to VLSI design which 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.

III. Conclusions and Suggestions for Future Work.

It seems clear, therefore, that precise specification of abstract interfaces for VLSI design modules is a crucial problem, a problem whose solution could enable the pursuit of useful research in a number of areas related to design change management. Based on this conclusion, I anticipate conducting follow-on doctoral dissertation research which reports on the development and demonstration of a new method for abstract interface specification of VLSI designs. I will further show how automated tools can efficiently support the new specification method, making its use perceptibly cost-effective for designers.

Acknowledgment. Important contributions to the conclusions reached above were made by each of the participants in this research exploration: the principal investigators, Frederick P. Brooks, Jr. and Peter Calingaert; Vernon L. Chi; Lynne C. Duncan; Lenwood S. Heath; Kye S. Hedlund; Roy McGuffin; John Poulton; Connie U. Smith; and Bernard I. Witt. Gerrit A. Blaauw, Kathryn Britton, and Henry Fuchs provided numerous helpful suggestions and comments.

Bibliography

[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.

[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.

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

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

[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.

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

[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.

[Chmu82]. Chmura, L.J. and D.M. Weiss, "The A-7E Software Requirements Document: Three Years of Change Data." *Naval Research Laboratory Memorandum Report 4938,* November 8, 1982.

[Gros83]. 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.

[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.

[Heat83]. Heath, L.S., "Criteria for Modular Decomposition in VLSI Layout." University of North Carolina at Chapel Hill Department of Computer Science Working Paper, June 24, 1983.

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

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

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

[Parn72]. Parnas, D.L., "On the Criteria to be Used in Decomposing Systems into Modules." *Communications of the ACM 5,* 12 (December 1972), pp. 1053-1058.

[Parn76]. Parnas, D.L., "On the Design and Development of Program Families." *IEEE Transactions on Software Engineering SE-2,* 1 (March 1976), pp. 1-9.

[Rade82]. Rader, J., ed., "Proceedings of the IEEE Computer Society VLSI and Software Engineering Workshop," Port Chester, NY, October 4-6, 1982.

[Séqu83]. Séquin, C. H., "Managing VLSI Complexity: An Outlook." *Proceedings of the IEEE 71,* 1 (January 1983), pp. 149-166.

[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.

[Weis84]. Weissman, C.A., private communication, June 1984.

[Wern83b]. Werner, J., "The Moving Target." *VLSI Design 4,* 2 (March/April 1983), p. 14.

[Wirt71]. Wirth, N., "Program Development by Stepwise Refinement." *Communications of the ACM 14,* 4 (April 1971), pp. 221-227.

Embedding Planar Graphs in Seven Pages
(Extended Abstract)

Lenny Heath*

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27514

ABSTRACT

This paper describes an algorithm for embedding any planar graph in a book of at most seven pages. This algorithm improves upon the result of Buss and Shor [1], which gave a nine page embedding. The algorithm uses a different level structure for planar graphs than did the algorithm of Buss and Shor to achieve the seven page result.

*Research supported in part by Semiconductor Research Corporation grant 41258.

1. Introduction

A *book embedding* of a graph is an ordering of its vertices along the spine of a book (i.e., linearly) and an assignment of each edge of the graph to a page of the book so that if two edges are on the same page of the book, then they do not intersect. This research investigates the problem of embedding any planar graph in a book of few pages. The minimum number of pages within which a graph can be book-embedded is called its *pagenumber*. It is known that the graphs of pagenumber one are exactly the outerplanar graphs and that the graphs of pagenumber two are exactly the subhamiltonian planar graphs (planar graphs that can be edge-augmented to have hamiltonian circuits, yet remain planar) [2]. Edges outside the circuit are placed in one page, edges inside the circuit are placed in a second page; edges on the circuit may be placed in either page. Since there are maximal planar graphs that are not hamiltonian, there are planar graphs with pagenumber at least three. Buss and Shor [1] give an algorithm that embeds any planar graph in nine pages. This paper gives an algorithm of quite different construction which embeds any planar graph in seven pages.

Clearly, if an algorithm is found to embed any triangulated planar graph, then any planar graph can be embedded; triangulate the planar graph, embed the triangulated graph and finally remove edges added by triangulation. We consider here a slightly broader class of planar graphs, inner-triangulated planar graphs. All triangulated planar graphs are inner-triangulated. Also, the planar embedding of a graph is fixed except for local modifications explicitly carried out by the algorithm. The main result of this research is that any inner-triangulated planar graph (and hence any planar graph) can be embedded in a seven page book.

The book embedding problem consists of two parts. First, the vertices of the graph are linearly ordered (in fact, the ordering can be circular, so the vertices are placed on a circle). Second, the edges are assigned to pages such that no two edges on the same page intersect. When the vertices are placed on a circle, the edges of the graph become chords of the circle and the second part becomes to color the chords with few colors so that no two intersecting chords are colored alike. Thus, a book embedding algorithm chooses the ordering of the vertices first so as to make possible the assignment of edges to a few pages later (though in reality the two parts may be done in parallel). The algorithm presented here starts with an assignment of vertices and edges to levels so that components of the graph are adjacent if and only if they are on the same or adjoining levels of the graph. It then uses the level structure to place the vertices on a circle and finally assign edges to pages.

2. Definitions

An *inner-triangulated planar graph* is a connected undirected graph (without loops or multiple edges) that can be embedded in the plane so that the exterior face is bounded by a cycle and any interior face is bounded by a triangle. Clearly, any triangulated planar graph is inner-triangulated. Henceforth, whenever a graph is given to be inner-triangulated, a planar embedding of the above nature is assumed given.

Let G=(V,E) be an inner-triangulated planar graph. We now define levels for the vertices and edges of G. The definitions are iteratively derived. V_0, the set of *level 0 vertices,* contains exactly the vertices on the exterior face of G. E_0, the set of *level 0 edges,* contains exactly the edges on the exterior face of G. G_0, the *level 0 subgraph* of G, equals (V_0, E_0). Hence, by the definition of inner-triangulated planar graph, G_0 is just the bounding cycle of the exterior face of G.

Now, suppose we have defined V_{k-1}, E_{k-1}, and G_{k-1} for $k \geq 1$. V_k, the set of *level k vertices,* is the subset of $V - \bigcup_{i=0}^{k-1} V_i$ consisting of vertices adjacent to vertices in V_{k-1}. An edge (v_1, v_2) is in E_k, the set of *level k edges,* if $v_1, v_2 \in V_k$ and there exists $v_3 \in V_{k-1}$ such that (v_1, v_2, v_3) is a face of G. G_k, the *level k subgraph* of G, is (V_k, E_k). Clearly, V is the disjoint union of all non-empty V_k, $k \geq 0$.

X_k, the set of *level k cross edges,* contains exactly the edges between level k vertices that are not in E_k. $B_{k,k+1}$, the set of *level k to k+1 binding edges,* contains an edge (v_1, v_2) if v_1 is a level k vertex and v_2 is a level k+1 vertex or vice versa. Clearly, E is the disjoint union of all the non-empty E_k,X_k and $B_{k,k+1}$, for $k \geq 0$.

The set E_k can be further partitioned. C_k, the set of *level k cycle edges,* contains edge (v_1, v_2) if (v_1, v_2) is in a cycle of G_k. N_k, the set of *level k non-cycle edges,* equals $E_k - C_k$.

A vertex $v \in V_k$ is a *pinch vertex* if v is contained in more than one cycle of G_k. A pinch vertex v is said to be *separated* from a cycle C of G_k in a book embedding if v is in C and vertices not in C are placed between v and the remaining vertices of C.

We need the following result (stated without proof) to continue definitions.

Theorem 2.1. Let $k \geq 0$. If C is a cycle of G_k, then any edge interior to C is in either X_j, $j \geq k$, $B_{j,j+1}$, $j \geq k$ or E_j, $j > k$ and any vertex interior to C is in V_j, $j > k$. Also, if (v_1, v_2) is an edge in X_j, $j \geq k$, $B_{j,j+1}$, $j \geq k$ or E_j, $j > k$, then (v_1, v_2) is contained in the interior of some cycle of G_k. If $v \in V_j$, $j > k$, then v is contained in the interior of some cycle of G_k.

Proof is by induction. *square*

If C is a cycle of G_k, then the subgraph of G consisting of C and its interior, $G|C$ (read G *restricted* to C), is an inner-triangulated planar graph containing only edges and vertices at levels $j, j \geq k$. Hence, subsets of the previously defined leveled sets which are restrictions to the interior of C can be defined. For example, $V_{j|C}, j \geq k$, consists of all vertices of V_j which are on C or interior to C. $B_{j,j+1|C}$, $j \geq k$, consists of all edges of $B_{j,j+1}$ which are interior to C.

Note that since $G|C$ is inner-triangulated, results for G translate to results for $G|C$.

Let v_i be a vertex of a cycle C of G_k. Let v_{i-1} and v_{i+1} be the vertices of C adjacent to v_i. Then there is a path in $G|C$ from v_{i-1} to v_{i+1} that includes only vertices adjacent to v_i. Define P_{v_i} to be this path.

3. Preliminary results

The following results are stated without proof. They are important to the understanding of the leveled structure of G. The important ideas are that G_k is a boundary in the sense of a closed curve in the plane and that cycles of G_k are disjoint, except perhaps for a shared pinch vertex.

Theorem 3.1. If C is a cycle of G_k and $X_{k|C}$ is empty, then $G_{k+1|C}$ is connected.

Theorem 3.2. Suppose $v_1, v_2 \in N_k$. Then there exist exactly two distinct vertices $v_3, v_4 \in V_{k-1}$ such that (v_1, v_2, v_3) and (v_1, v_2, v_4) are faces of G. (Implication in the other direction works also.)

Theorem 3.3. Suppose $(v_1, v_2) \in C_k$. Then (v_1, v_2) is in only one cycle of G_k.

4. Levels without cycles

If G has only two non-empty levels and level 1 contains no cycle, then we can show that G is subhamiltonian in a special sense. The cyclic order of vertices in G_0 can be preserved in the hamiltonian circuit. The algorithm for obtaining this hamiltonian circuit motivates the vertex ordering of our main algorithm in section 5.

Theorem 4.1. If G_1 contains no cycles, then G_1 is a forest and no level greater than 1 is non-empty.

Proof: Obvious. *square*

Theorem 4.2. If G_1 contains no cycles and is connected, then G_1 is a tree.

Proof: Obvious. *square*

Theorem 4.3. If G_1 contains no cycles and X_0 is empty, then G_1 is connected, and there exists a hamiltonian cycle H for G such that the vertices of G_0 appear in H in the same order as they do in the cycle G_0.

Proof: The algorithm for constructing H sheds some light on the forthcoming algorithm for book embedding. The proof that G_1 is connected is omitted here. By Theorem 4.2, G_1 is a tree. Assume the vertices of G_0 are labeled in cyclic order $v_1, \cdots, v_m$. Suppose this order to be clockwise. Since X_0 is empty, all vertices interior to the path P_{v_i} are in V_1. We may assume G_1 is non-empty. Then for each v_i, there exists some vertex of G_1 adjacent to v_i. Let P_{v_1} be given by $v_m, u_1, \cdots, u_n, v_2$ where $u_j \in V_1$.

Start H at v_i. Go to u_1 and follow P_{v_1} to v_2. Now for $2 \leq i < m$, when v_i is reached, go to the first vertex of P_{v_i} that has yet to be visited by H and if that vertex is not v_{i+1}, follow P_{v_i} to v_{i+1}. When v_m is reached, return to v_1 to complete H. Every vertex of G is contained in H. Following P_{v_i} to v_{i+1} cannot encounter a vertex already in H because then G_1 would contain a cycle (the proof is omitted here). Clearly H contains V_0 in the order $v_1, \cdots, v_m$. *square*

Theorem 4.4. If G_1 contains no cycles, then G is subhamiltonian where the order of the vertices of G_0 in the hamiltonian cycle will be preserved.

Proof: Assume G_1 not empty. If X_0 is empty, apply theorem 4.3. If X_0 is non-empty, re-embed the X_0 edges outside G_0 (this can be done by outerplanarity) and triangulate the interior of G_0 such that no new cross edges are introduced. Then, theorem 4.3 applies to G_0 and its interior. Edges of X_0 will necessarily be exterior to the hamiltonian circuit. *square*

5. Algorithm for book embedding

The algorithm for embedding an arbitrary inner-triangulated planar graph in a book draws on the ideas of the previous section. However, the embedding will be seen in a circle for convenience. The algorithm lays out the vertices of G level by level. It starts by laying the vertices of G_0 out around the circle in cycle order. At each level k, the algorithm will already have placed the vertices of levels 0 through k and it seeks to place the vertices of level k + 1. It does so by taking each cycle in G_k and laying out the level k + 1 vertices interior to the cycle separately. To make the layout of each cycle independent of all others, the vertices of each cycle of G_k should be laid out consecutively, without intervening vertices and in cycle order. This will always be possible except when there are pinch vertices. In that case, the algorithm places the vertices of the cycle in cycle order and possible leaves some level k - 1 vertices between a single pinch vertex and the rest of the cycle vertices. It is important that at most one vertex of a cycle be separated from the rest of the vertices.

Cross edges can cause problems for the layout. When a cycle has cross edges, the technique of the previous section will be applied. The cross edges are re-embedded exterior to the cycle and the interior of the cycle is triangulated without adding new cross edges. Edges added during triangulation can be deleted at the very end of the algorithm.

Let C be a cycle of G_k. The algorithm places the vertices of $G_{k+1|C}$ by finding a cycle through the vertices of C and $V_{k+1|C}$. The cycle will not really be a hamiltonian circuit since some edges of $E_{k+1|C}$ must be deleted and some edges added to complete the cycle. In particular, for any cycle of $G_{k+1|C}$, exactly one edge will be deleted. Suppose (v_1, v_2) is the edge in $E_{k+1|C}$ to be deleted and let $v_3 \in V_{k|C}$ such that (v_1, v_2, v_3) is a face of G. Then when (v_1, v_2) is deleted, edges from v_3 to all remaining vertices of the level k + 1 cycle can be added to the planar embedding (if the interior of the cycle is ignored). Then the algorithm traverses the entire level k + 1 cycle from v_1 around to v_2 and is able to proceed to the next vertex.

The algorithm assumes the existence of a pool of seven pages, $S = \{s_1, \cdots, s_7\}$. Place the vertices of G_0 around the circle in cycle order. Choose an arbitrary vertex $w \in V_0$ and invoke LAYOUT$(G_0, w, s_1, \{s_5, s_6, s_7\})$.

LAYOUT is a procedure called with four parameters. The first parameter is a cycle C in some G_k. The second is a vertex v of C. The third is the page s on which edges incident to v are to be placed. The fourth is a set T of three pages that may already be assigned to edges incident to C and hence may not immediately be reused. LAYOUT can assume that the vertices of C are already placed in cycle order and that there is an arc of the circle within which all vertices of C reside and no other vertices reside except that there may be some vertices of the level k - 1 cycle that contains C between v and the other vertices of C. Also, if C contains a pinch vertex that was separated from C at the next outer level, then v is that pinch vertex of C. The algorithm will place vertices of $G_{k+1|C}$ within the arc strictly containing vertices of C.

The following is a sketch of the LAYOUT procedure. It is invoked as LAYOUT(C, v, s, T). If $V_{k+1|C}$ is empty, then set H equal to C and proceed to the next paragraph. Move all cross edges of C outside of C and triangulate the interior of C so that no new cross edges are introduced and so that $G_{k+1|C}$ is connected. Let C be defined by vertices $v = v_1, \cdots, v_m$. Let $\{r_1, r_2, r_3\} = S - (T \bigcup \{s\})$. Let P_{v_1} be defined by $v_m, u_1, \cdots, u_n, v_2$. Go from v_1 to u_1 and along P_{v_1} until a cycle edge (u_j, u_{j+1}) is encountered. Let K be the cycle of $G_{k+1|C}$ containing (u_j, u_{j+1}). Associate page r_1 with K. Delete (u_j, u_{j+1}) and add edges from v_1 to every vertex of the cycle, rerouting when necessary. Go from u_j to u_{j+1} along the cycle, then go to the next un-encountered u_i or v_2 as necessary. Invoke LAYOUT $(k, r_2, \{s, r_1, r_3\})$. Continue around C. At each v_i, find the first unencountered edge of P_{v_i}. If it is a cycle edge (u_j, u_{j+1}) and u_j is already encountered, do as before except when going around the cycle u_j will be skipped. Let K be the cycle of $G_{k+1|C}$

containing (u_j, u_{j+1}). Let K' be the unique cycle of $G_{k+1|C}$ which contains u_j and which u_j was not separated from. K' is hence the first cycle containing u_j which was laid out. If r_1 is associated with K', invoke LAYOUT$(K, r_1, \{s, r_2, r_3\})$ and associate r_2 with K. If r_2 is associated with K', invoke LAYOUT$(K, r_2, \{s, r_1, r_3\})$ and associate r_2 with K.

When LAYOUT has returned to v_m, a hamiltonian circuit H has been defined containing vertices of C in cyclic order and all vertices in $V_{k+1|C}$. Assign page s to all edges incident to v. Now assign pages s and r_3 to edges of C and edges in $B_{k,k+1|C}$ and $X_{k|C}$ as indicated by the hamiltonian circuit H. This completes LAYOUT.

Whenever LAYOUT is entered, its assumptions are met. In particular, the following assumption holds: when LAYOUT(C, v, s, T) is invoked, any vertices between v and the remainder of C have edges on only five of the seven pages. All edges in G that are incident to vertices of C are assigned to one of the five pages in $T \bigcup \{s, r_3\}$. Either page r_1 or r_2 is assigned to all edges from a pinch vertex at the next level that may intersect those edges, thus avoiding conflict with the five pages incident to vertices of C.

It is now clear that the reason the algorithm must use seven pages is the possible existence of pinch vertices for cycles of C. If G has no pinch vertices, the algorithm can easily be modified to use only four pages.

6. Conclusion

This paper has presented an algorithm for embedding any planar graph in seven pages. The algorithm can be shown to have time performance $O(n^2)$, so that it is efficient. Seven pages is a two-page improvement over the previously best known upper bound of nine [1] and uses a new approach specifically developed and analyzed for this problem. Future research might consist of closing the gap between the best current lower bound of three pages and this new upper bound of seven pages. The author conjectures that four pages is a lower bound for pagenumber of planar graphs based solely on the investigation that resulted in this new algorithm, as he has no example that he knows requires more than three pages.

7. Acknowledgement

The author expresses his appreciation to Arnold L. Rosenberg of Duke University for suggesting the problem and much encouragement. The author also appreciates the support provided by the Semiconductor Research Corporation.

8. References

[1] J. Buss and P. Shor, "On the pagenumber of planar graphs," *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing,* 1984, pp. 98-100.

[2] F. R. K. Chung, F. T. Leighton and A. L. Rosenberg, "A graph layout problem with applications to VLSI design," in preparation.