TECHNICAL REPORT NO. 289

A System for Digital Design Derivation

by

Steven D. Johnson and Bhaskar Bose

August 1989

COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY

Bloomington, Indiana 47405-4101

TECHNICAL REPORT NO. 289

A System for Digital Design Derivation

by

Steven D. Johnson and Bhaskar Bose

August 1989

COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY

Bloomington, Indiana 47405-4101

TECHNICAL REPORT NO. 289

A System for Digital Design Derivation

by

Steven D. Johnson and Bhaskar Bose

August 1989

COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY

Bloomington, Indiana 47405-4101

A System for Digital Design Derivation*

Steven D. Johnson and Bhaskar Bose
Computer Science Department
Indiana University

1. Introduction

The Digital Design Derivation system (DDD) is a facility for synthesizing digital implementations of higher level algorithmic specifications. It is being developed to explore an approach to design based on the algebraic manipulation of functional expressions. This summary reviews the motives for DDD's development, outlines the synthesis process, and reviews experience with the system.

DDD is closer in character to a mechanical theorem prover than to a hardware compiler [2], except that it manipulates design descriptions rather than assertions about them. Intuitively, it supports reasoning from a specification toward an implementation, while a verification system does the opposite. However, derivation and verification are complementary aspects of design, not conflicting approaches to it. A principal reason for developing DDD is to provide a framework for exploring the integration of synthesis with formal verification.

Engineering by algebraic refinement is fundamental to the discipline of *functional programming*. Results in that area lead to constructions of *sequential systems*, which provide a formal characterization of hardware [5]. DDD is an experimental implementation of the algebra, which has evolved through a series of concrete design exercises.

One benefit of automating design "in a logic" is descriptive abstraction, with its potential adaptability to higher levels of description. DDD reflects no particular technology, architecture, or problem class. The research goal is to characterize these

---

*This research was supported, in part, by the National Science Foundation under grants numbered MIP8707067 and DCR8521497.

Summary to be presented at the ACM/IEEE Workshop on High-Level Synthesis, October, 1989.

aspects through tactical use of the algebra. Thus, the challenge of the approach lies in managing the abstraction permitted by the formal system.

2. Background

DDD is a collection of transformations for manipulating expressions toward physically meaningful forms. The synthesis process is called "derivation" to emphasize that these expressions are dialects of a single modeling language—higher order functional notation. A *derivation* is a sequence of transformations,

$$\mathcal{D}_0 \xrightarrow{T_0} \mathcal{D}_1 \xrightarrow{T_1} \cdots \xrightarrow{T_{k-1}} \mathcal{D}_k$$

Source expression $\mathcal{D}_0$ is sometimes called a *specification* and target expression $\mathcal{D}_k$ an *implementation*. However, it is more accurate to say that a design is described by sequence $\langle T_0, \ldots, T_{k-1} \rangle$ applied to $\mathcal{D}_0$, since the sequence states much of the design intent.

Figure 1 shows the flow of information in DDD; nodes stand for expressions and edges stand for transformations.

The system is open for integration of other functional programming tools, and it produces a collection of boolean system descriptions for input to available logic synthesis tools. Derivation is driven by script of transformation commands composed by the engineer. A derivation typically has three phases, addressing behavioral, structural, and physical aspects of design. An algorithm in functional form is translated into a *system description*. Next, a series of *factorizations* imposes a conceptual architecture. Third, DDD incorporates concrete representations and restructures the description for realization.

3.1. Control Synthesis

Specifications are given in terms of an arbitrary *basis type*, or vocabulary of constants, operations and tests. They are systems of function definitions composed of *conditional expressions*—a language of simple applicative terms, recursive invocations, *if-then-else* expressions and *case* expressions. DDD currently requires that specifications be *iterative*—the class of definition schemata associated with finite state machine descriptions. Thus, DDD is a *register transfer synthesizer* in the vocabulary of Keutzer and Wolf [4], but because the level of description is not fixed, the notion of 'register' is abstract.

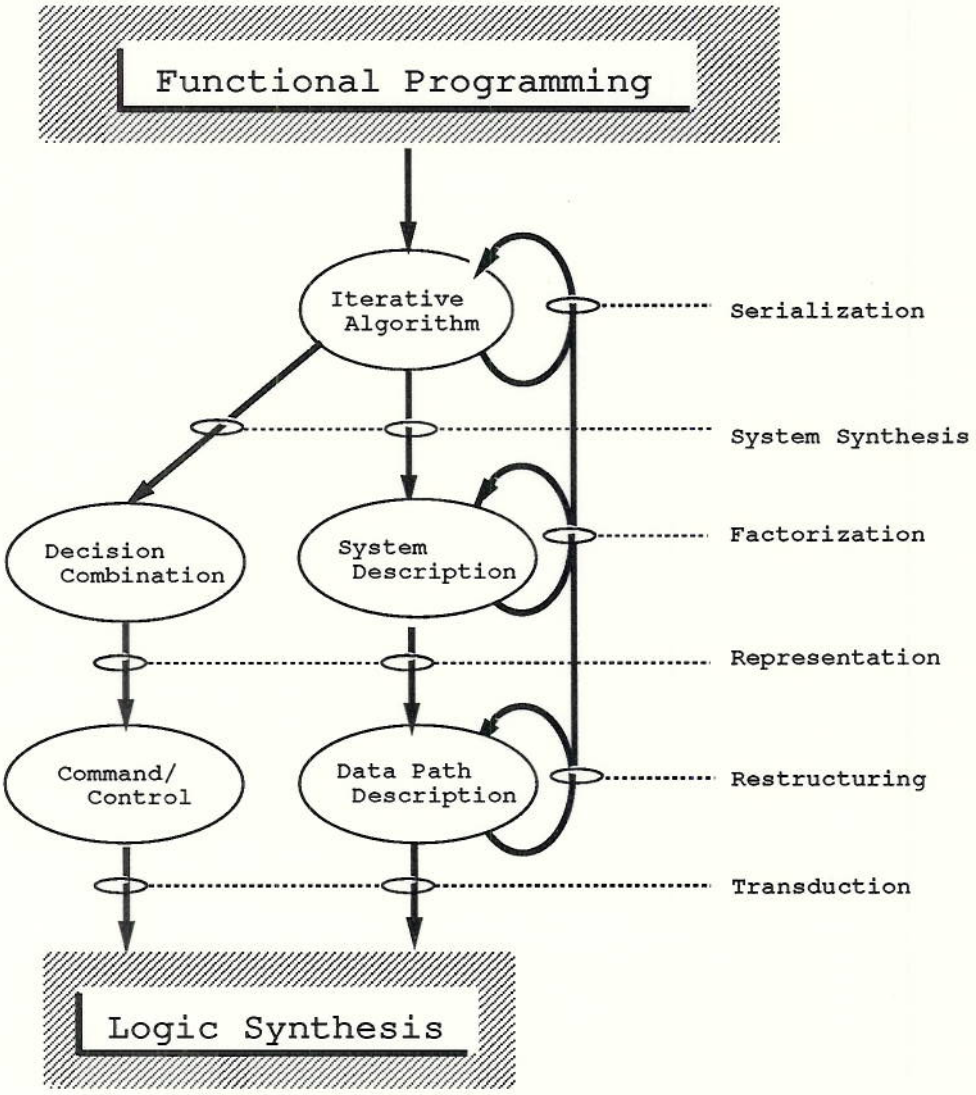
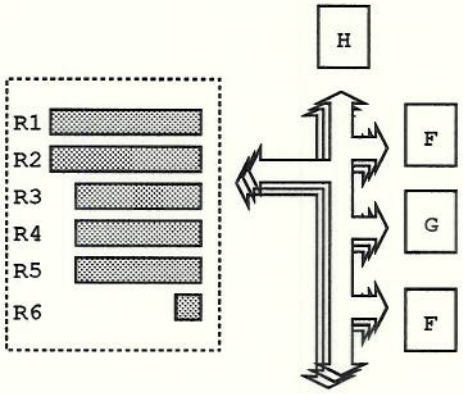


FIGURE 1 Design derivation in DDD.

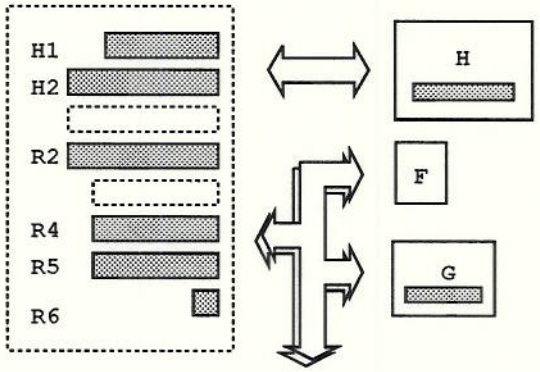
DDD builds *sequential system descriptions* according to a theoretical construction in [5]. The initial system does not necessarily describe implementable architecture. The diagram to the right expresses certain aspects of its structure for the following discussion. Register entities (R1, R2, *etc.*) are connected to various basis operations (H, F, and G in the figure). A synthesized *decision combination* (not shown) governs data movement. Typically, there is a high degree of parallelism in data transfer. In other words, the initial system, though correct, would not ordinarily be reduced to a realization. It may be described at too high a level and the architecture is often unrealistic.



3.2. Manipulating Logical Organization

The central phase of derivation imposes modularity by merging expressions and encapsulating subsystems. The transformations are called *system factorizations.*

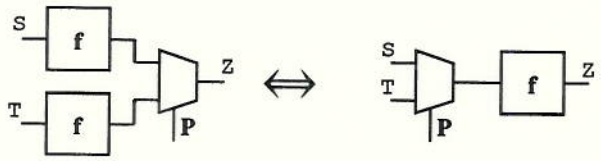
The diagram to the right illustrates some of the effects of factoring. Multiple instances of the common subexpression F are reduced to a single component. The register R3 is incorporated in the function G; an example is encapsulating an integer register and an incrementer as a counter. An important third use of factorization corresponds to information hiding. A type abstraction, such as memory or queue, is encapsulated as a process entity. Such transformations often introduce of new registers—in the diagram, H1 and H2 result when R1 is incorporated with H. For example, a factored memory would leave residual address and content registers.



The factorization algebra is based on a distributive law of selection. As it is usually applied, the law "pushes" selectors through functions. If *sel* is a two-way selector,

$$sel(P, f(S), f(T)) \quad \equiv \quad f(sel(P, S, T))$$

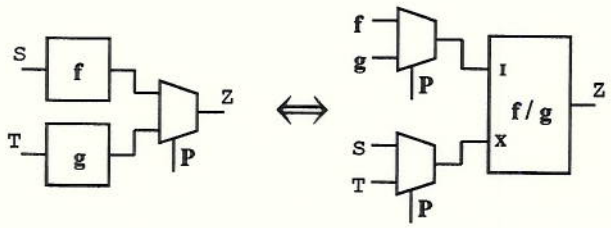
The effect on a diagram looks like



Factorization hinges on a more general law, a basic identity for higher order expressions that asserts distributivity over functions as well as arguments:

$$sel(P, f(S), g(T)) \quad \equiv \quad (sel(P, f, g))(sel(P, S, T))$$

In DDD, distribution introduces a component which performs either f or g depending on an instruction, f or g:

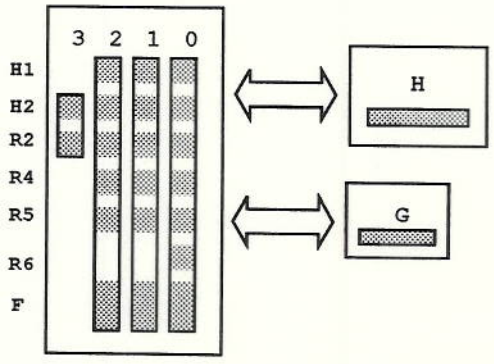


An example is the assignment of various arithmetic operations to an ALU. However, such transformations also apply to arbitrary collections of terms, and are often used to develop hierarchy in a description.

Failures in factorization induce refinements to control, which are accomplished by *serializing* transformations on the specification (See Figure 1).

3.3. Developing Physical Organization

The final phase of a derivation introduces (more) concrete representations for the values and operations of the basis type. At the binary representation level, DDD generates a collection of boolean subsystems for implementing the design. The decomposition may be orthogonal to the logical hierarchy of a structural description. A common example, depicted to the right, is restructuring the data path into bit slices. Registers and certain combinational functions are projected to a single bit in each component of the physical



organization. The algebraic characterization of this phase involves a third class of transformations, which perform pervasive changes to the design description. It is a task of the DDD system to sustain correctness during the restructuring.

The boolean subsystems are put into existing logic synthesis facilities, which perform optimizations and assemble realizations. Working prototypes have been built in PLA and PLD/MSI technologies. So far, the exercises have been data path oriented designs; future experimentation will address other problem classes.

3. Experience and Research Directions

Our experience with DDD is evidence that a 'formal' approach to design can develop into a practical tool. The most revealing exercise is a garbage collector project reported in [1, 6]. The design involves a significant hierarchy of type abstractions: a *Heap*, implemented by parameterized type, *Memory*(*Address*, *Content*), represented by $\mathsf{RAM}(\mathsf{Bit}^n, \mathsf{Bit}^m)$. Two derivations from a single specification were targeted to a highly parallel PLD prototype, and a highly serial PLA prototype. The specification text, executing with a production LISP system, generated test patterns for a switch level simulation of the VLSI realization.

The system has also been applied to a mechanically verified microprocessor description. Starting from Hunt's FM8501 specification [3], DDD synthesized an implementation comparable to the one proved in Boyer-Moore Logic. DDD was able to manage a specification that was composed for the purpose of proof rather than implementation. The exercise demonstrates the utility of an algebraic approach in dealing with logical constructs.

The DDD system is open ended. Existing program transformation methods are planned for the specification level. We shall continue to develop derivation tactics for other target technologies. Finally, much remains to be learned about design at higher levels of description.

References

[1] C. David Boyer and Steven D. Johnson, "Using the Digital Design Derivation System: Case Study of a VLSI Garbage Collector Implementation," *Proceedings of the IFIP WG 10.2 Ninth International Symposium on Computer Hardware Description Languages* (1989), to appear.

[2] DANIEL D. GAJSKI, *Silicon Compilation,* Addison-Wesley, Reading, 1987.

[3] WARREN A. HUNT, JR., *FM8501: A Verified Microprocessor,* Ph.D. dissertation, also published as Technical Report 47 (December, 1985), Institute of Computing Science, The University of Texas at Austin, 1985.

[4] KURT KEUTZER AND WAYNE WOLF, "Anatomy of a Hardware Compiler," *Proceedings of the ACM SIGPLAN '88 Conference on Programming Design and Implementation,* Atlanta, 1988.

[5] STEVEN D. JOHNSON, *Synthesis of Digital Designs from Recursion Equations,* The MIT Press, Cambridge, 1984.

[6] STEVEN D. JOHNSON, BHASKAR BOSE, AND C. DAVID BOYER, "A Tactical Framework for Digital Design," in *VLSI Specification, Verification and Synthesis,* (eds.) Graham Birtwistle and P.A. Subramanyam, Kluwer Academic Publishers, 1987, 349–384 (proceedings of the 1987 Calgary Hardware Verification Workshop).

[7] STEVEN D. JOHNSON, BHASKAR BOSE AND ROBERT W. WEHRMEISTER, "On the Interplay of Hardware Derivation and Hardware Verification: Experiments with the FM8501 Microprocessor Description," submitted.