DDD-FM9001: Derivation of a Verified Microprocessor †

An Exercise in Integrating Verification with Formal Derivation

Bhaskar Bose and Steven D. Johnson

Indiana University, Bloomington IN 47405, USA

Abstract. The *DDD-FM9001* is a 32-bit general purpose microprocessor *formally derived* directly from Hunt's mechanically verified *Nqthm FM9001 microprocessor* specification. The exercise was part of a project to construct an implementation of the FM9001 by applying the *DDD design derivation system* to the Nqthm FM9001 specification. The main thesis of this work maintains that *derivation* and *verification* represent interdependent facets of design and must be integrated if formal methods are to support the natural analytical and generative reasoning that takes place in engineering practice. In this paper we describe the continuation of previous work in which the DDD system was applied to Hunt's *FM8501 specification.* This paper describes the derivation of the DDD-FM9001 and compares the derived architecture and hardware realization with that of the FM9001 in an effort to better understand the interplay between derivation and verification.

1 Introduction

Derivation and *verification* are interdependent facets of design which reflect alternate modes of reasoning in the design space. Derivation reflects a design perspective where algebra is used to correctly transform a *specification* into an *implementation.* Verification reflects a design perspective where a proof establishes that an implementation satisfies its specification.

As a continuation in our efforts to understand how derivation and verification are interrelated, we applied the DDD system [11, 12, 14], a transformation system which implements a basic design algebra of equivalence preserving transformations for circuit derivation, to Hunt's FM9001 specification [10]. Hunt mechanically verified the FM9001 in the Nqthm theorem prover [5]. A set of

† Research reported herein was supported, in part, by NSF: The National Science Foundation, under grants numbered DCR 85-21497, MIP 87-07067 and MIP 89-21842, and by NASA: The National Aeronautics and Space Administration under grant number NGT-50861.

transformations were applied to decompose and reorganize the design. A controller, next-state function, and datapath were derived. Complex components, such as the memory, register file, ALU, incrementor, and decrementor, were isolated using DDD's abstraction mechanisms. Technology dependent, highly optimized implementations of the arithmetic components were engineered and verified against their respective abstract specifications. The derived and arithmetic components were implemented in an ACTEL FPGA (field programmable gate array) [1]. The memory and register file were implemented using SRAM components. The result of this experiment was a derived FM9001.

This work extends the experimentation on the interplay between derivation and verification that was reported in [15]. Previous work applied the DDD system to Hunt's FM8501 description [9]. Results of this work exposed the need to take a broader view of formal reasoning in design. The DDD/FM8501 experiment illustrated how alternative modes of reasoning could be applied to a single design. The work showed how the massive restructuring involved at lower levels of abstraction could be implemented more easily by derivation, and how the inventive aspects of a design could be isolated for verification.

The DDD/FM9001 exercise extends the previous work on the FM8501 in three ways. First the derivation was upgraded in conjunction with Hunt's refinements to representation. Second, much more of the algebra was mechanized; in fact, the entire ACTEL gate-level hardware description was generated either by mechanical derivation or verified using boolean equivalence methods [2, 6]. Finally, the DDD-FM9001 was realized in hardware.

2 Experiences from the FM8501 Experiment

In the FM8501 proof [9], Hunt established an equivalence relation between specifications of an abstract programmer's model, called `soft`, and an implementation, a hardware interpreter model, called `big-machine`. DDD was applied to both the abstract programmer's model, `soft`, and the hardware interpreter model, `big-machine`. The derived architecture for `soft` was quite close to Hunt's implementation, however, it did not contain certain key registers such as those associated with the memory protocol. These registers were not expected to arise in the derivation since they did not exist in the original specification of `soft`. In fact, this difference highlighted an essential aspect of Hunt's proof. Hunt's proof established an equivalence relation between a functional model of memory with that of a process model of memory.

In the second derivation exercise, DDD was applied to the hardware interpreter model, `big-machine`, to guide a top-down expansion of the design. Unlike Hunt's approach, in which a bottom-up expansion of `big-machine` resulted in over 11 million gates (reduced to 1,789 gates with the identification of like terms), algebraic manipulations were used to unfold, decompose, and re-

structure `big-machine` while containing the size of the expansion. The derived architecture was identical to Hunt's block diagram.

Experiences with the FM8501 experiment developed two central ideas. The derivation of `soft` exposed elements of an implementation that could not be derived from a specification. These elements reflected isolated components of a design that must be proved. The `big-machine` derivation illustrated the need for transformational algebra to restructure and decompose a design in order to manage the logical and physical organization necessary to construct a realization targeted to a particular technology. Our experiences with the FM8501 gave us insight into the FM9001 derivation.

3 The FM9001 Microprocessor

The FM9001 [10] is a 32-bit microprocessor representing the third generation processor description defined by Hunt and mechanically verified using the Nqthm theorem prover [5]. The proof establishes an equivalence relationship between four levels of specifications ranging from an *abstract programmer's model interpreter* to a *netlist.*

The highest level of specification (Figure 3) is a collection of six recursive functions defining an instruction level interpreter of the FM9001. This level is referred to as the programmer's model. The composition of these six functions define an abstract behavioral description: the FM9001 interpreter. The state of the machine is defined by a memory, `mem`, a register file, `regs`, four flags, `c,v,n,z`, a program counter, `pc-reg`, an instruction register, `ins`, the operand registers, `operand-a` and `operand-b`, and an address calculation register, `b-address`. The programmer's model is defined to have 32-bit addressing, 16 general purpose registers, 5 addressing modes, a 16 function ALU, and a conditional assignment statement. A block diagram of the architecture is illustrated in Figure 4.

4 Description of the DDD-FM9001 Derivation

The DDD-FM9001 derivation was done interactively with the DDD system. The transformations were applied to the specification, developing a derivation path through the design space satisfying an intended set of design constraints. An initial script was developed to reflect a sequence of commands to the DDD system. Several derivation paths were explored and the derivation refined. The final derivation script includes 30 coarsely grained commands to the derivation system. The complete script consists of 1,000 lines with a total of 38,000 characters.

Figure 1 illustrates the derivation path from Hunt's *FM9001 Specification* to the *DDD-FM9001 Realization.* Transformations on the descriptions are shown

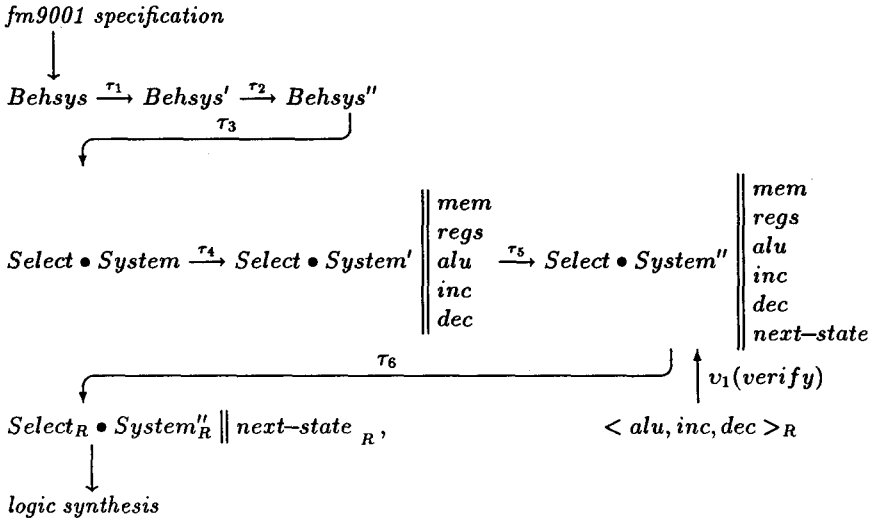

Fig. 1. DDD-FM9001 Derivation Path

as labeled arcs, $< \tau_1 ... \tau_6, \upsilon_1 >$, where τ_n denotes the application of a transformation and υ_r denotes verification. The diagram is intended to characterize the distinct phases of the derivation. A class of transformations called, *behavioral transformations*, were applied to the initial specification in order to achieve a proper scheduling of operations. Once a suitable behavioral description was derived, DDD constructed an abstract system description, composed of a decision combinator, *Select*, representing control, and a structural component, *System*, representing an initial estimation of architecture. A second class of transformations, called *structural transformations*, imposed a logical organization on the design. In this phase of the derivation, the structural description was refined to an architecture. A sequence of factorization steps were applied encapsulating complex signals as co-processes. The || operator denotes a communicating subsystem. Implementations for the arithmetic components were hand designed and mechanically verified with respect to the factored components. A third class of transformations introduced a lower-level representation producing a hierarchy of boolean subsystems. A final gate-level description was input to the ACTEL logic synthesis tool. A detailed account of the derivation follows.

Transformations on the Behavioral Specification

τ_1 : An initial set of transformations were applied to the FM9001 behavioral specification. Some auxiliary definitions were expanded and conditional expressions manipulated in order to rearrange the specification. For example,

conditional expressions were moved outside the function invocation. The function invocation `fm9001-alu-operation` is transformed in order to move the conditional outside the call.

```
(fm9001-alu-operation
 ... (if (reg-direct-p mode-b)
         (read-mem (rn-b ins) regs)
         (read-mem b-address mem)) ...)
```

is transformed to

```
(if (reg-direct-p mode-b)
    (fm9001-alu-operation ... (read-mem (rn-b ins) regs) ...)
    (fm9001-alu-operation ... (read-mem b-address mem) ...))
```

In DDD, this has the effect of moving the decision point of a predicate into the synthesized control component. If the conditional had been left within the function invocation, it would be implemented in the architecture.

τ_2 : In a process analogous to *scheduling* in high-level synthesis [16], the DDD system was guided through a series of folding and unfolding transformations in order to achieve a desired scheduling of operations. This was discussed in [15] but not mechanized at the time. The goal was to reduce the inherent parallelism in the original specification. In this phase, called *serialization*, a function call is replaced by a sequence of function calls, whose composition is equivalent to the original term. New registers may also be added if necessary to store intermediate results.

For example, the `fm9001-fetch` function in Figure 3 was serialized to impose an ordering on the memory and register file read/write operations. A new function was created to represent the addition of a state. A temporary register, `tmp`, was also created to hold the intermediate value being read from the register file. In this example, further serialization steps will be necessary to serialize the two remaining read/write operations on the register file and memory in `fm9001-fetch_1`.

```
...
(defn fm9001-fetch (regs flags mem pc-reg)
  (let ((tmp (read-mem pc-reg regs)))
    (fm9001-fetch_1 regs flags mem pc-reg tmp)))

(defn fm9001-fetch_1 (regs flags mem pc-reg tmp)
  (let ((ins (read-mem tmp mem)))
    (let ((pc+1 (v-inc tmp)))
      (let ((new-regs (write-mem pc-reg regs pc+1)))
        (fm9001-operand-a new-regs flags mem ins)))))
...
```

Seven serialization steps were necessary to produce a design in which abstract operations on memory and the register file were restricted to at most one memory access per state. The operations were also serialized to insure that accesses to memory and the register file could be multiplexed. A single temporary register and five new states were added to the design.

System Synthesis

τ_3 : The next step involves a series of transformations which decompose the behavioral specification into a control abstraction and a structural component. Details of this transformation is reported in [4, 11, 14]. The transformations are completely automatic and require no user guidance. At this point in the derivation, the structural component represents an initial estimation of architecture and a logical organization must be imposed on the design.

Structure to Architecture

τ_4 : In the next phase, DDD's abstraction mechanisms were used to transform the design into a reasonable description of functional components for implementation. Operations were encapsulated in modules and complex components such as the memory, register file, ALU, incrementor, and decrementor were factored from the description [13]. Figure 2 diagrams this decomposition.

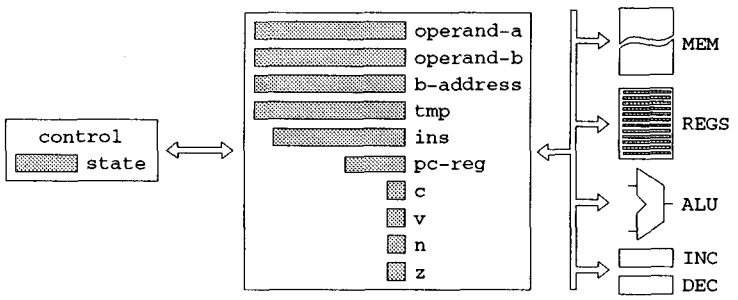

Fig. 2. DDD-FM9001 Logical Organization

The isolation of components into abstract modules provides a mechanism by which individual components can be verified. It is this process which is exploited in the derivation that allows the integration of verification.

$\tau 5$: As an optimization, the data bus for the memory and register file were merged. The next-state function was derived.

Architecture to Physical Organization

$\tau 6$: Next, concrete representations were introduced for the constants and operations of the ground type. A collection of boolean subsystems were then generated. This set of transformations represent a massive restructuring of the design. The transformations could not have been done manually. For example, the datapath description increased by a factor of approximately 11 by going from a description of 8,000 characters, to a description of approximately 90,000 characters.

At this stage in the derivation, the design had been decomposed into a controller, a datapath, and abstract modules for memory, the register file, ALU, incrementor, and decrementor. The controller and datapath could be assembled directly into hardware. The memory and register file could be implemented with SRAMs. However, we wanted to implement an engineered solution for the ALU, incrementor, and decrementor. These modules needed to be verified to preserve the integrity of the design.

Boolean Verification

v_1 : The verification of the arithmetic components in the DDD-FM9001 reduced to verifying a boolean term, constructed from the FM9001 abstract ALU specification, was equivalent to a hand designed multiplexor design tuned for the ACTEL FPGA architecture. Details of the verification are reported in [3]. Algebraic techniques were used to construct a boolean term from the abstract ALU specification. Symbolic evaluation, where base operators are extended to return symbolic values and symbols are introduced as input values in place of real data objects [8] , was applied to both the abstract ALU specification and the ACTEL multiplexor implementation.

For example, to construct a boolean term from the expression `(v-and operand-a operand-b)`, where `operand-a` and `operand-b` denote 32-bit vectors (ie: `[operand-a_0, operand-a_1, ..., operand-a_31]`), and the definition for `v-and` given below:

```
(defn v-and (x y)
  (if (nlistp x) nil
      (cons (b-and (car x) (car y))
            (v-and (cdr x) (cdr y)))))
```

Symbolically evaluating `(v-and operand-a operand-b)` returns

```
(b-and (b-and operand-a_0 operand-b_0)
       (b-and (b-and operand-a_1 operand-b_1)
              ...
              (b-and operand-a_31 operand-b_31)))
```

Technology dependent, highly optimized implementations of the ALU, incrementor, and decrementor were engineered to map to the ACTEL FPGA architecture. The designs were then verified in COSMOS [7] using Boolean verification [2, 6] against their respective abstract specifications. Figure 5 shows what aspect of the design was verified and what was derived. The shaded area denotes the components that were verified. The unshaded components, except for the register file, R, were derived.

4.1 DDD-FM9001 Realization

The final result of the derivation was a realization of the DDD-FM9001 specification. The DDD-FM9001 realization is implemented as a chip set incorporating an ACTEL FPGA and a SRAM module for the register file. The register file was made external due to the area limitations of the the ACTEL FPGA technology and speed considerations. The design was targeted to FPGAs since they provided a cost effective rapid prototyping solution.

5 Observations and Future Work

The DDD-FM9001 and FM9001 are equivalent in the sense that they share a common abstract specification. Both processors define a formal relationship between the abstract programmers model specification and its implementation. The DDD-FM9001 was derived and the FM9001 was verified. In terms of hardware realizations, the machines execute the same instruction set and exhibit the same state to state behavior for each instruction cycle. At the end of each instruction, the contents of the register file, and memory are equivalent. However, the implementations differ in several key respects.

Some of the quantitative differences between the FM9001 verification and DDD-FM9001 derivation are given in the table below. The table provides some basis for analysis, however the interesting comparisons relate to the differences in architecture.

	FM9001 Verification	DDD-FM9001 Derivation
Script entries	2957 entries	1000 lines
Execution time	4hrs (Sparc 2)	30min (Sparc 2)
Netlist	91K characters 2215 lines	69K characters 1178 lines
I/O Pins	95 32 bi-directional	91 32 bi-directional

The block diagrams for the FM9001 and DDD-FM9001 (Figure 4 & 5) show similar yet distinct architectures. This is not a surprise since the goal of the

```
(defn fm9001-intr (state oracle)
  (if (nlistp oracle) state
      (fm9001-intr (fm9001-step state (car oracle)) (cdr oracle))))

(defn fm9001-step (state pc-reg)
  (let ((p-state (car state)) (mem (cadr state)))
    (fm9001-fetch (regs p-state) (flags p-state) mem pc-reg)))

(defn fm9001-fetch (regs flags mem pc-reg)
  (let ((pc (read-mem pc-reg regs)))
    (let ((ins (read-mem pc mem)))
      (let ((pc+1 (v-inc pc)))
        (let ((new-regs (write-mem pc-reg regs pc+1)))
          (fm9001-operand-a new-regs flags mem ins))))))

(defn fm9001-operand-a (regs flags mem ins)
  (let ((a-immediate-p (a-immediate-p ins))
        (a-immediate (sign-extend (a-immediate ins) 32))
        (mode-a (mode-a ins)) (rn-a (rn-a ins)))
    (let ((reg (read-mem rn-a regs)))
      (let ((reg- (v-dec reg)) (reg+ (v-inc reg)))
        (let ((operand-a (if* a-immediate-p a-immediate
                              (if* (reg-direct-p mode-a) reg
                                   (if* (pre-dec-p mode-a) (read-mem reg- mem)
                                        (read-mem reg mem))))))
          (let ((new-regs (if* a-immediate-p regs
                               (if* (pre-dec-p mode-a) (write-mem rn-a regs reg-)
                                    (if* (post-inc-p mode-a) (write-mem rn-a regs reg+) regs)))))
            (fm9001-operand-b new-regs flags mem ins operand-a)))))))

(defn fm9001-operand-b (regs flags mem ins operand-a)
  (let ((mode-b (mode-b ins)) (rn-b (rn-b ins)))
    (let ((reg (read-mem rn-b regs)))
      (let ((reg- (v-dec reg)) (reg+ (v-inc reg)))
        (let ((b-address (if* (pre-dec-p mode-b) reg- reg)))
          (let ((operand-b (if* (reg-direct-p mode-b) reg (read-mem b-address mem)))
                (new-regs (if* (pre-dec-p mode-b) (write-mem rn-b regs reg-)
                               (if* (post-inc-p mode-b) (write-mem rn-b regs reg+) regs))))
            (fm9001-alu-operation new-regs flags mem ins operand-a operand-b b-address)))))))

(defn fm9001-alu-operation (regs flags mem ins operand-a operand-b b-address)
  (let ((op-code (op-code ins)) (store-cc (store-cc ins)) (set-flags (set-flags ins))
        (mode-b (mode-b ins)) (rn-b (rn-b ins)))
    (let ((cvzbv (v-alu (c-flag flags) operand-a operand-b op-code))
          (storep (store-resultp store-cc flags)))
      (let ((bv (bv cvzbv)))
        (let ((new-regs (if* (and* storep (reg-direct-p mode-b)) (write-mem rn-b regs bv) regs))
              (new-flags (update-flags flags set-flags cvzbv))
              (new-mem   (if* (and* storep (not* (reg-direct-p mode-b)))
                              (write-mem b-address mem bv) mem)))
          (list (list new-regs new-flags) new-mem))))))
```

Fig. 3. Hunt's FM9001 Programmer's Model Specification

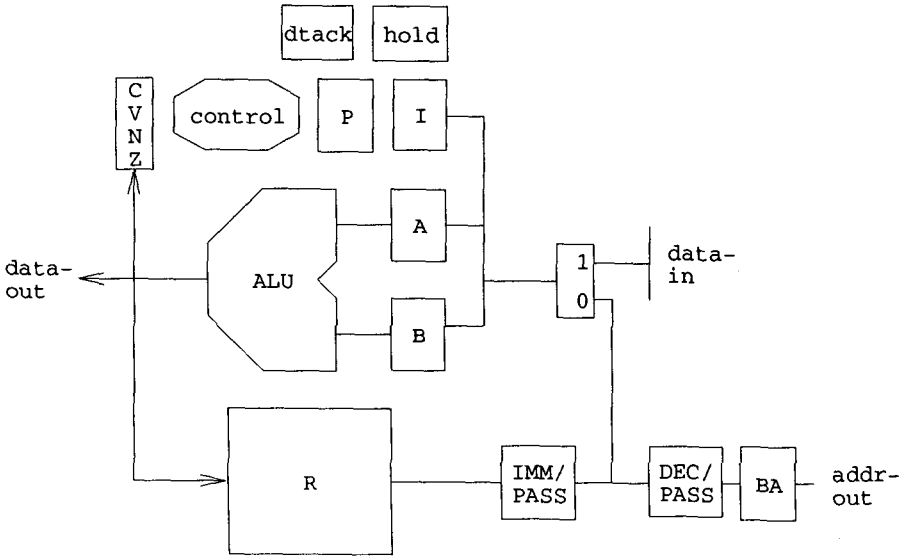

Fig. 4. FM9001 Block Diagram

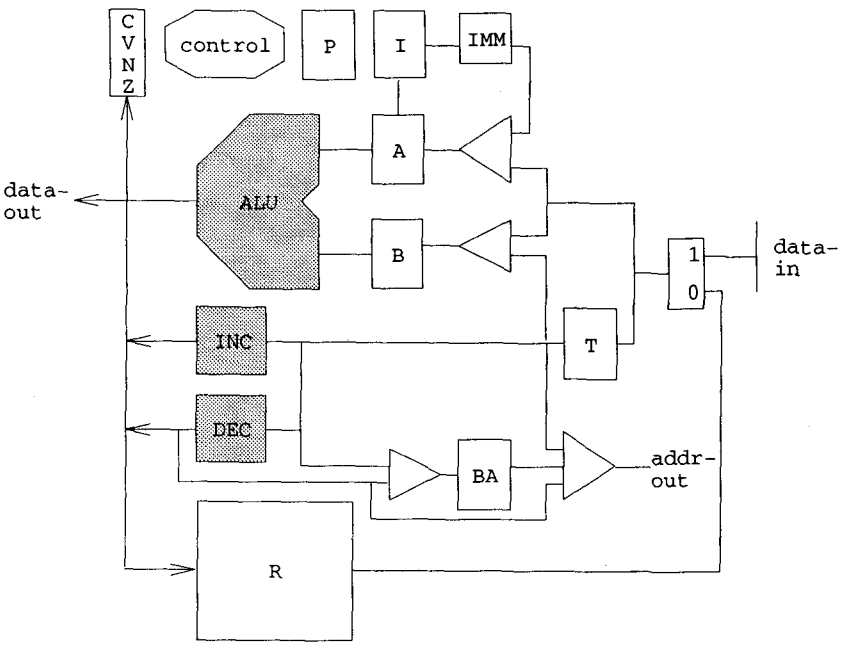

Fig. 5. DDD-FM9001 Block Diagram

derivation was not to achieve the identical implementation as the FM9001, but to derive an implementation that preserved the behavior of the initial specification.

The DDD-FM9001 has a temporary register, T, which does not exist in the FM9001. The register was a result of a design decision made by the designer during the serialization phase of the derivation. A temporary register was created in order to store an intermediate value. Liveness analysis of the registers would allow us to eliminate this register by using an existing register instead.

The DDD-FM9001 implements both an incrementor and decrementor. The FM9001 implements only a decrementor. The FM9001 ALU is used to implement the increment operation. Serialization can be used to make sure the ALU, incrementor and decrementor operations are scheduled so that they do not occur in the same clock cycle. This would allow us to factor the arithmetic operations as a single component and verify it against an implementation.

The significant difference between the FM9001 and the DDD-FM9001 is the absence of the `dtack` and `hold` registers, and the scan path, in the derived design. The `dtack` signal relates to the change in the model of memory from a functional abstraction to a process abstraction. This difference isolates an aspect of the verification of the FM9001. They could not be derived since they did not exist in the original specification. We have developed a formalism for process decomposition to address this very issue[17], however, at the time of this derivation exercise, the function was not integrated with DDD.

At present, a new derivation is under way. In this derivation, the behavioral specification will be serialized such that the incrementor, decrementor and ALU operations will be implemented with a single arithmetic component. In addition, live variable analysis will be integrated with the serializer so that we can use the `operand-a` and `operand-b` registers instead of the temporary register. Also, sequential abstraction will be incorporated in order to derive an implementation with the appropriate synchronization protocols necessary to interface with a DRAM implementation of memory.

As we begin to understand the interplay between derivation and verification, we can move towards a design framework which supports both forms of reasoning. In this paper we discuss a small aspect on how derivation and verification interrelate. In the DDD-FM9001 experiment, boolean verification was incorporated into the derivation path providing a formal mechanism to verify the arithmetic components of the design. However, we are interested in the broader issues relating to how multiple reasoning systems in a peer relationship interact in design. We feel that these issues, at the core, are subtle and will require further investigation.

References

1. Actel Corporation, Sunnyvale, CA. *ACT Family Field Programmable Gate Array Databook*, 1991.
2. S. B. Akers. Binary Decision Diagrams. In *IEEE Transactions on Computers*, volume C-27, pages 509–516, June 1978.
3. Bhaskar Bose, Steven D. Johnson, and Shyamsundar Pullela. Integrating boolean verification with formal derivation. In *Proceedings of the IFIP Conference on Hardware Description Languages and their applications (CHDL)*, 1993.
4. C. David Boyer and Steven D. Johnson. Using the digital design derivation system: Case study of a VLSI garbage collector. In J. Darringer and F. Ramming, editors, *Proceedings of the IFIP WG 10.2 Ninth International Symposium on Computer Hardware Description Languages (CHDL)*, Amsterdam, 1989. Elsevier.
5. Robert S. Boyer and J. Struther Moore. *A Computational Logic Handbook.* Academic Press, New York, 1988.
6. R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. In *IEEE Transactions on Computers*, volume C-35, pages 677–691, August 1986.
7. R. Bryant. COSMOS: A Compiled Simulator for MOS Circuits. In *Proceedings of the* 24^{th} *Design Automation Conference*, 1987.
8. John A. Darringer. The application of program verification techniques to hardware verification. In *Proceedings of the* 16^{th} *Design Automation Conference*, June 1979.
9. Jr. Hunt, Warren A. *FM8501: A Verified Microprocessor.* PhD thesis, The University of Texas at Austin, 1985. Also published as Technical Report 47 (December, 1985).
10. Warren A. Hunt. A formal HDL and its use in the FM9001 verification. In C.A.R. Hoare and M.J.C. Gordon, editors, *Mechanized Reasoning in Hardware Design.* Prentice-Hall, 1992.
11. S. D. Johnson and Bhaskar Bose. A system for mechanized digital design derivation. In Subramanyam, editor, *Proceedings of ACM International Workshop on Formal Methods in VLSI Design.* January 1991.
12. Steven D. Johnson. *Synthesis of Digital Designs from Recursion Equations.* The MIT Press, Cambridge, 1984.
13. Steven D. Johnson. Manipulating logical organization with system factorizations. In M. Leeser and G. Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, pages 260–281. Proceedings of Mathematical Sciences Institute Workshop, Cornell University, Springer-Verlag, July 1989.
14. Steven D. Johnson, Bhaskar Bose, and C. David Boyer. A tactical framework for digital design. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 349–383. Kluwer Academic Publishers, Boston, 1988.
15. Steven D. Johnson, Robert M. Wehrmeister, and Bhaskar Bose. On the interplay of synthesis and verification: Experiments with the FM8501 processor description. In Luc Claesen, editor, *Applied Formal Methods For Correct VLSI Design*, pages 385–404, Amsterdam, Netherlands, 1989. IMEC, Elsevier.
16. Michael C. McFarland, Alice C. Parker, and Raul Camposano. Tutorial on high-level synthesis. In *25th ACM/IEEE Design Automation Conference*, pages 330–336, 1988.
17. Kamlesh Rath and Steven D. Johnson. Toward a basis for protocol specification and process decomposition. In *Proceedings of the IFIP Conference on Hardware Description Languages and their applications (CHDL)*, 1993.