Technical Report No. 544

Formal Derivation of a Scheme Computer

by Steven D. Johnson

September 2000

First posted electronically in March 1997 at
`http://www.cs.indiana.edu/hmg/schemachine.ps.Z`

Computer Science Department
Indiana University
Bloomington, Indiana 47405-4101

Formal Derivation of a Scheme Computer*

Steven D. Johnson

September 12, 2000

Abstract

This report describes a proposed project involving the formal derivation and system-level verification of a computer for executing compiled Scheme.

1 Introduction

Figure 1 shows a prototype computer for compiled Scheme [8] programs. The *Schemachine*'s memory subsystem maintains a symbolic processing heap, including binary, string, and array objects plus varieties of list cells and numeric representations. Its processing unit is tailored for Scheme run-time objects, but does not include floating point hardware. Four of the five major components, roughly 95% of the gate network, were derived from executable algorithmic specifications using a mechanized transformation system. We developed the *Digital Design Derivation (DDD)* system to explore algebraic programming methodology applied to digital system design.

Language implementation, from fully abstract semantic specification to a native-code implementation is a standard exercise in programming language research; however, formalized synthesis of a working hardware implementation is a new achievement.

> PROJECT GOAL: *A number of outstanding verification problems remain before the entire system in Figure 1 could be considered correct in a formal sense. Our primary objectives are, first,*

*This research was supported, in part, by the National Science Foundation under grants numbered MIP9208745 and MIP9610358.

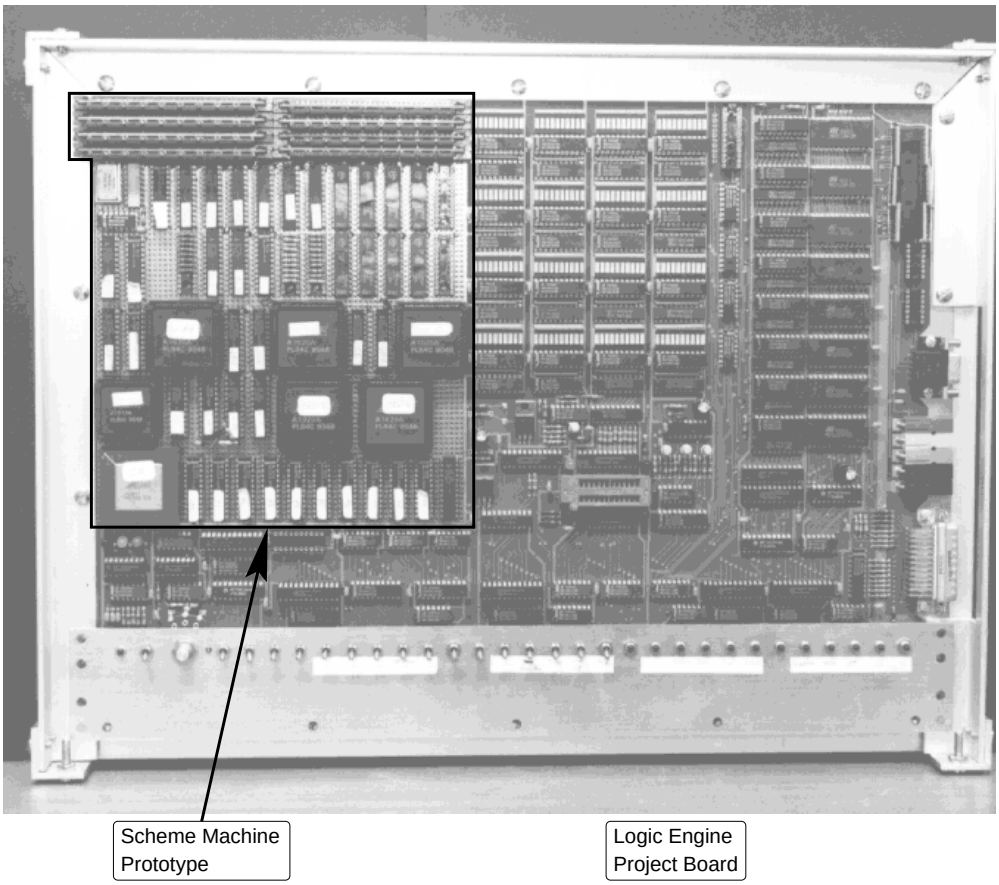


Figure 1: Scheme computer prototype

to solve these remaining problems to obtain a complete implementation proof, and, second, to reestablish links to the compiler derivation research through which the Schemachine's specification originated.

Sections 2.1, 2.2, and 2.3 describe four specific tasks for completing the Schemachine verification. A variety of reasoning tools are needed to complete this work, but the reasoning framework, that is, the organizing system in which reasoning subtasks are are integrated, is DDD.

An underlying topic of this research is *heterogenious reasoning*, exploring the interaction of different formal systems applied to a common design problem. DDD implements a specialized algebra for simply typed systems of function definitions (representing transition systems) and networks of streams (representing architectures). Its transformations enable the user to construct efficient hardware implementations while preserving behavioral equivalence. We use theorem provers, model checkers, and other reasoning tools to verify partial properties, such as the correctness of arithmetic representations, the noninterference of processes, or the validity of refinements.

Practitioners often speak of the difference between proving implementations and proving specifications. From that perspective, the first project objective is an implementation proof for a full system consisting of a storage manager, garbage collector, processing unit, and primitive memory. As such, it represents an advance in higher level verification research, which until now has focused on individual system components, such as the processor.

"Specification correctness," our second objective, is inherited from a larger body of research establishing a mathematical basis for accepting Schemachine, in composition with a compiler, as a valid Scheme executor. Both DDD and the Schemachine reflect research in programming methodology and compiler derivation by Wand, Friedman, Wise, Clinger, Haynes, among others, starting in the early 1980s. In fact, the Schemachine's processor is almost the same as the virtual machine found in the textbook describing this methodology (Section 2.1). Similarly, there is a strong connection to VLISP, a mathematically rigorous treatment of Scheme by Guttman, Wand, Ramsdell, and others [17]. Fuller accounts of these relationships are in Sections 1.2, 3.1, and **??**, the last of which is the required summary of recent NSF supported results.

1.1 Contributions of the research.

Generally, this research advances our understanding of system design and produces tools and techniques reflecting that understanding. The results lay the groundwork for more rigorous engineering practice, and more effective training. We are interested both in broadening the scope and lowering the cost of these practices.

In whole, this is a substantially more ambitious verification study than has been attempted before. Although the immediate tasks deal with hardware implementation, the methodology encompasses "programming" in a much broader sense, transcending distictinctions between hardware and software. The need for such a broadening is more evident than ever in system synthesis, in reconfigurable technologies, in co-design and embedded software, and in distributed programming.

We develop a particular formalism for a particular technology with the understanding that a practical methodology must integrate several forms of reasoning over a wide range of technologies. Challenging case studies are fundamental to this science. It is encouraging that more and more of the challenges are now arising through industrial experience. However, while practical studies extend the reach of existing verification technology, studies like the Schemachine push the horizon of what we consider to be a verifiable object. Language implementation is one kind of stress test for higher abstractions of behavior, data, interface, and structure, and this project examines of how those are ultimately reflected in hardware.

Second, this project contributes to a large body of work toward rigorous treatments of Scheme. Scheme is an extremely advanced programming/modeling language. It is a strong candidate for design-critical applications. Schemachine is a proof in principle, and very nearly in fact, that a high level lanuage can be secured in dedicated hardware using formal methods with a reasonable level of effort.

Finally, this project will produce useful artifacts for general purpose symbolic processing. Although Schemachine's processor is primarily an academic exercise (at the moment) its memory system is novel, robust, and adaptable to other architecures, especially for embedded software. Naturally, we are looking at JAVA as a potential target since it relies on a fully garbage collected heap (Section 3.2). If we can cultivate commercial partners, we will also contemplate a DDD derivation of the JAVA virtual machine [23].

1.2 Background and related research

The origins of this research lie in programming methodology work of the early 1980s. Wand's compiler derivation results [31, 30] were particularly influential. Very briefly, Wand's technique involves a factorization of language semantics into a recursive compiler and an iterative machine component, together with systematic techniques for introducing concrete representations for more abstract data objects. This algebraic style of reasoning is not just a way to develop compilers, however. It is the essence of a programming methodology by Friedman, Wise, Haynes, Wand, and many others as presented in the text book *Essentials of Programming Languages* [14].

Starting around 1980, Johnson began adapting the methodology for the construction of formal hardware models [19, 18, 20]. As early as 1983, a specific goal was outlined to extend Wand's compiler work by deriving a hardware executor from the machine component [19, ch. 5]. Johnson asked whether the language theorist's notion of a virtual machine was adequate for obtaining a reasonable hardware system. He found a significant gap between the two, and embarked on a research program to close it.

By 1986, a first version of the DDD transformation system was used to derive a garbage collector and realize it in programmed logic arrays [21, 5]. The collection algorithm was taken from, and tested against, an experimental Scheme implementation due to Clinger [10]. This was the initial step in the Schemachine project.

Clinger's language work was influential in other ways. His Scheme compiler proof [9] gave us an early perspective on heterogeneous reasoning. That proof presented as a retrospective argument about a given compiled bytecode interpreter. In contrast, Wand's style felt more constructive in that an implementation results from a series of refinement steps. However, these contrasting forms of argument are entirely compatable. *The Proof* is just an explanation of the invention process, which is neither purely analytic nor purely generative. This notion of interacting reasoning styles is not reflected in current reasoning tools. It is a central topic of our research.

There was a good deal of related work in language-specific architecture during the mid 1980s. A group at MITRE corporation built a combinator-reduction machine and were in the process of formalizing it in M-EVES when funding ran out [26]. Birtwistle's group at Calgary proved a VLSI implementation of Landin's SECD machine in HOL [16]. Wehrmeister independently used DDD to derive a working SECD computer [32]. Both

SECD machines machines included storage managment for a simple binary-cell memory. Wehrmeister derived these as separate co-processes; Birtwistle used a common data path.

In 1989-90, we built the high-performance memory for a more realistic Scheme implementation. We re-targeted the earlier collector to this dual-ported memory object. A processor based on Clinger's byte-code interpreter [10] was derived and nearly realized to hardware, but the student working on it withdrew from graduate school. In 1992, Burger finished a processor derivation [7], starting this time from a virtual machine sketch in [14, Ch. 12], which by that time also incorporated Clinger's ideas.

The MITRE group, eventually joining forces with Wand, produced a Scheme implementation called VLISP, explicating the compiler/machine derivation with extreme mathematical, albeit not mechanical, rigor [17]. The VLISP virtual machine [28] is quite close in form and level of abastraction to our Schemachine specification. We believe DDD could be used to derive a hardware implementation; see Section 3.1.

Computational Logic's *Short Stack* is currently the most advanced study of a verified programming system [15, 1]. It is an interpreter hierarchy consisting of a microprocessor, loader, primitive operating system, assembler, a compiler for a simple imperative language, and a verification condition generator for that language. In a subsequent report, Flatau describes the partial proof of a heap based applicative language in the same hierarchy [12]. The languages and memory models in Short Stack are more basic than either VLISP's or Schemachine's, but their correcntess proofs are fully machine checked.

The cornerstone of CLI's short stack is Hunt's FM9001 microprocessor [6], which actually exists as a VLSI gate array. Bose applied the DDD system to the Hunt's FM9001 specification, deriving a functionally equivalent piece of hardware [4, 3, 2]. This work contrasts derivational and deductive styles of verification, demonstrating that, in practice, one wants both.

Schemachine's relationship to VLISP is similar to that of FM9001 and Short Stack. One difference is that Schemachine is a whole system architecture, including a non-trivial memory subsystem. Hence, for Schemachine, the outstanding verification problems have to do with with system-level coordination, rather than device-level correctness. FM9001 is just the processing component of a system, although it does have a specified memory interface, and more of Short Stack's operating system is contained in the proof. Also, the FM9001 hardware is built on a single chip and is six times faster. But

these are transitory distinctions, since both studies are ongoing.

2 Research problems and topics

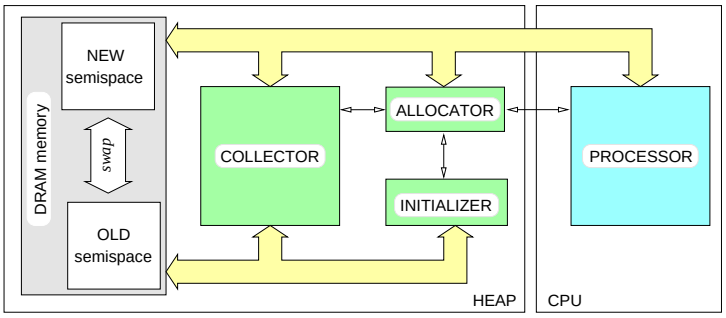


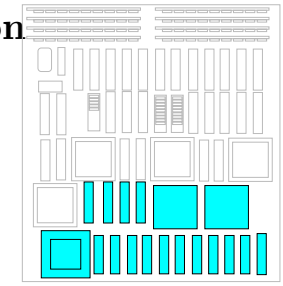
Figure 2: Schemachine architecture

The conceptual architecture of Schemachine is shown in Figure 2. There are five principal components. A manually designed, dual ported primitive memory providing two heap semi-spaces, uses standard DRAM SIMMS and PLD devices. The other four components are obtained by DDD derivations and realized in FPGA technology. A heap manager consists of collector, initializer, and allocator processes. The cpu implements a virtual machine for compiled Scheme.

Primitive memory is tailored to raise performance of the stop-and-copy collector algorithm by supporting parallel read-write operations on the two semi-spaces. In addition, the initializer runs in parallel with the cpu, sweeping the inactive semi-space between garbage collections.

The prototype pictured in Figure 1 is actually a *hardware emulator*; its gate network projected into a number of inexpensive and reusable devices. Physically, the design decomposes into a processor, heap manager, and primitive memory interface, all of which could now fit on a single IC. But these should instead be three distinct devices; we will be looking at caching strategies for collector and cpu that will consume a lot of chip area.

2.1 Refinement of the CPU derivation



TASK A *is to extend the DDD formal system to close the remaining gap between the intuitive machine specification sketched in Figure 3 and the architecturally biased DDD specification sketched in Figure 4.*

The Summary of Chapter 12 in the textbook *Essentials of Programming Languages* outlines a method for factoring a language definition into a recursive compiler and an interpreter of compiled actions. Figure 3 shows a fragment of one form the *machine* component can take: a tail-recursive function with formal parameters representing the machine's internal state. (*cf.* `eval-code` in Fig. 12.4.4 of [14]). It is is a reasonable hardware specification, but current DDD is not quite capable of reducing Figure 3 to a reasonable architecture.

Figure 4 shows the corresponding DDD specification fragment actually used for Schemachine. Owing to the inexperience of the specifier, it is considerably more detailed than would have been necessary, but the problem it illustrates is representative. The `if-inst` instruction leaves the accumulator unchanged, but transitions to `fetch` contain

$t \equiv$ `(make-cite (obj.tag acc) (alu-out (alu a+0 (obj.ptr acc) ?  ff))`

for the next-state value of `acc`. This term anticipates an architecture that always routes `acc` through the processor's ALU, adding `0` in this instance. DDD does not have enough term-rewriting capablility to justify replacing t with `acc` even though to do so is perfectly valid.

Figure 4 is essentially an unfolding of Figure 3 into a system of about 170 microstates. DDD does have fold/unfold capabilities, but even an experienced DDD user would have to start with around 10 macrostates. We want the formal derivation to start with Figure 3, or something even more abstract.

Most of the 170 microstates explicate individual primitive operations. Since these are ultimately allocated to arithmetic units, we should be able to encapsulate them earlier and avoid any intermediate expression as large as Figure 4. Of course, it is in the character of this research to repair such deficiencies, only to expose more of them at higher levels of specification. Even so, what is now permitted in DDD is substantially higher than that of

```
(define elc (lambda (acc val* env k code pc) ...
   (variant-case (vector-ref code pc)
      ...
      [lit-inst (datum)
         (elc datum val* env k code (add1 pc))]
      ...
      [if-instruction (else-loc)
         (if (true-value?  acc)
            (elc acc val* env k code (add1 pc))
            (elc acc val* env k code else-loc))]
      ...))
```

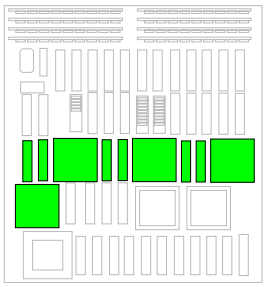
Figure 3: Fragment of the Scheme instruction interpreter [7]

```
(define scheme-machine
   (lambda (go interrupt halt)
   (letrec (
      ...
      [fetch (lambda (acc val* lex-env k code pc tag addr ie port mem cont)
         (if (halt)
             (save ...)
         (if (and (interrupt) ie)
             (malloc ...)
             (fetch-1 ...))))]
      ...
      [if-inst (lambda (acc val* lex-env k code pc tag addr ie port mem cont)
         (if (not (same-type?  acc <imm> <false>))
             (fetch (make-cite (obj.tag acc)
                               (alu-out (alu a+0 (obj.ptr acc) ?  ff)))
                    val* lex-env k code (add pc (c24 0) tt) tag
                    (add addr (c24 0) ff) ie port mem cont)
             (fetch (make-cite (obj.tag acc)
                               (alu-out (alu a+0 (obj.ptr acc) ?  ff)))
                    val* lex-env k code
                    (add (obj.ptr (memrd mem pc)) code tt) tag
                    (add addr (c24 0) ff) ie port mem cont)))]
      ...))))
```

Figure 4: Fragment of the Schemachine CPU specification

current behavioral synthesis languages, and we must continue to extend the gap, in order to justify the interactive character of the system.

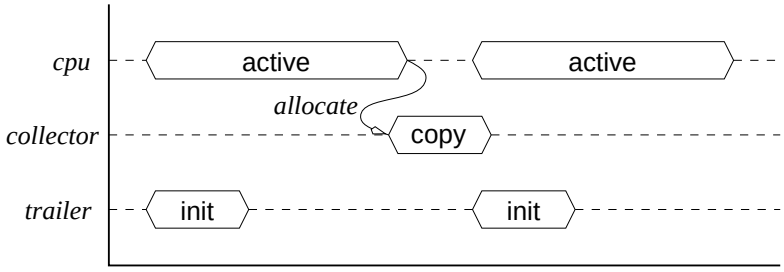
2.2 The heap manager



TASK B *is to enhance the stand-alone storage management subsystem for higher performance, resulting in a useful artifact for general purpose symbolic processing.*

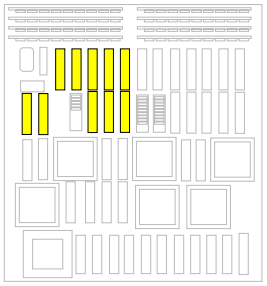
TASK C *is a validation that the major components of the Schemachine system are coherently synchronized.*

The heap manager consists of three processes. The allocator is the storage manager, mediating between the cpu and the collector, a stop-and-copy storage reclamation algorithm. The initilizer sweeps the inactive semispace in parallel with cpu, presetting an *invalid* bit in each cell. Pre-initialization has no overhead under normal cpu consumption patterns. A filter on the memory bus clears any invalid referent as it passes into the cpu. This, together with the property that the cpu cannot manufacture a pointer, guarantees the integrity of the heap, which in turn is a basis for assuring access security. Under this scheme, it is possible to have (prove) full security without memory protection hardware.



Noninterference among cpu, collector, and initializer is a coherence property. It may be automatically verifiable for appropriate reductions of collector, initializer, and allocator using finite-state methods [11]. We want a reasoning framework that accomodates reasoning with proxy specifications. Such a framewok needs mechanized transformation for reducing full specifications to abstracted processes that can be "model checked," as well as provisions for maintaining inherited properties in their instances.

2.3 The DRAM interface



TASK D *is to automatically verify the DRAM bus, for coherence, timing, and clocking.*

DRAMS are considerably more complicated than the simple memory models typical in verification studies. Ubiquitous in practice, DRAMS present a representative problem for high-level system specification and verification. Recently, the terms *interface specification* and *transaction modeling* have arisen for this problem class. Briefly, the topic is to develop formal rules to derive the nontrivial sequential interactions implementing abstract operations such as read and write.

The Schemachine is a good case study because a hierarchy of data abstractions is involved. DRAMs implement a dual-ported memory. The memory implements an array object. The array implements a heap. The heap implements both user data structures and the run-time objects (closures, continuations, etc.) for Scheme execution. The run-time objects implement byproducts of the compilation process.

Hierarchical data abstraction as developed in language research does not adequately address the temporal/behavioral aspects that dominates hardware system design. Our formalism for process decomposition is general enough to handle DRAMs [27] but not all these results have been incorporated in the DDD tool yet.

As Figure 5 shows, Schemachine's dual-ported memory model is realized by a single 32-bit bus in which access to the two semi-spaces are interleaved in both space and time. Certain frequently occuring pairs of memory accesses, can be done in 4/3 of a DRAM write cycle. They include a *read/write* step in the innermost cycle of collector and any cpu access in parallel with initilizer's cell marking. A semispace *swap* is accomplished by reassigning the fields.

A hierarchical clocking scheme implements the two levels of sequential behavior. The system clock ticks at the variable rate of DRAM memory protocol, enabled by a microclock that makes asynchronous refresh cycles transparent to the higher behavior. Although verification studies of RISC pipelines involve multilevel time scales [13] interacting clocks pose new problems in formalization.

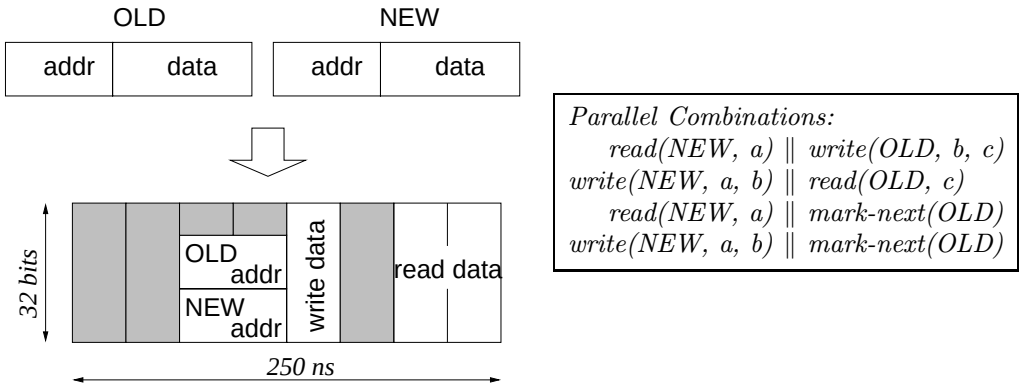


Figure 5: DRAM memory bus timing

3 Extending the research

3.1 Linking Schemachine and VLISP

> TASK E *is to investigate doing a hardware derivation of* VLISP*'s* FSBCM *machine model.*

VLISP is a mathematically rigorous implementation of Scheme, relating a fully abstract language semantics through several levels of interpretation to a particular compiler-machine decomposition. The final virtual-machine description, FSBCM, is very similar in to Schemachine's cpu specification. Figures 6 and 7 show fragments of the VLISP virtual machine description at levels corresponding to the DDD expressions in Figures 3 and 4, respectively. Schemachine's cpu is expressed as system of tail-recursive functions. FSBCM is expressed in an imperative style (*cf.* `eval-code` in Fig. 12.4.3 of [14]), but two modes of expression are interconvertable.

VLISP's machine is represented by a complex of 17 Scheme variables; seven are shown in Figure 7, and the rest are embedded in the memory model. Schemachine's cpu machine state is represented by 12 lambda parameters in Figure 4; there are seven more state elements in collector and allocator. The state size, level of abstraction, and complexity of actions are very similar in the two specifications. We therefore believe that DDD could reduce FSBCM to hardware. However, we anticipate problems in dealing with exceptions, which are apparently outside the FSBCM model because it is intended for native-code targets. There are potentially serious implications for heap integrity.

$$
\begin{array}{l}
\mathit{Fetch}(M)(ip, up, sp, s, h) \Rightarrow \mathit{Execute}(M)(ip, M(ip), up, sp, s, h) \\
\vdots \\
\mathit{Execute}(M)(ip, \textsf{jump-if-false}, up, sp, s, h) \\
\quad \Rightarrow \mathit{Fetch}(M)(ip', M(ip), sp - 1, s, h) \\
\qquad\qquad \text{where } ip' = \begin{cases} ip + 2 & \mathit{if}\ \mathit{truish}(s(sp)) \\ M(ip + 1) & \mathit{otherwise} \end{cases} \\
\vdots
\end{array}
$$

Figure 6: Fragment of VLISP's stored-code interpreter [29] (*cf.* Fig. 3).

```
;;;
;;; virtual machine state
;;;
   ...
   (define *template* unspecified)
   (define *codevector* *hp*)
   (define *offset* 0)
   (define *value* unspecified)
   (define *env* unspecified)
   (define *cont* unspecified)
   (define *spare1* unspecified)
   ...
;;;
;;; implementation of [jump-if-false m1 m2] instruction
;;;
   ...
   (if (= (value-ref) false)
       (set!  *offset* (+ 3 (compute-offset (current-instr-param 1)
                                            (current-instr-param 2))))
       (set!  *offset* (+ *offset* 3))
   ...
```

Figure 7: Fragment of VLISP's virtual machine, transcribed from [29]

3.2 Linking Schemachine and JAVA

> TASK F *is to initiate a long-term case study with the* JAVA *virtual machine, and derive variants of the heap subsystem for possible use in embedded design.*

Currently, there is considerable interest in verified JAVA core implementations, but the interested parties have already committed their cores to proprietary VLSI and are enlisting formal methodists for retrospective verification. Interest in derived implementations may develop, should JAVA reach status as a de facto standard. It is difficult to obtain a pre-determined implementation using DDD, but we may be able to target a general architecture and use automatic verification on its components. The JAVA virtual machine specification [23] is at roughly the same modeling level as our Scheme cpu, assuming some of its object-access instructions are factored into interface specifications. This as just the kind of decomposition problem, bridging software and hardware, that is of interest at this stage of our research.

At the same time, Schemachine's heap subsystem could readily be adapted to supporting JAVA execution. Given its security features this may be of interest to core providers. We will pursue this prospect if interest develops. We do plan to revise the collector for generation scavaging and to explore a page oriented caching scheme. Both these enhancement pose interesting formal challenges.

3.3 Summary

Tasks A (*p.* 8) and B (*p.* 10) enhance the DDD system to support higher design specifications and more complex architectural descriptions. Such enhancements can take the form of sophisticated transformations or better integration with other verification tools. This aspect is a continuation of our ongoing study of heterogenious reasoning in system design [24, 2, 25, 3, 22].

The essence of Task C (*p.* 10) is proving that the decomposition of Schemachine into five concurrent components preserves behavior and does not result in any interference pathologies. As noted earlier, we think this is the key issue facing design methodology today. Task D (*p.* 11), proving the physical memory interface, is clearly a candidate for automatic verification. Since it entails a shift in time granularity, how its proof plays into heterogenious reasoning is of central interest.

Task E (*p.* 12) re-integrates this work with research from which it originated. We are told that the VLISP effort is currently inactive, but hope this connection will rekindle interest for safety critical applciations. Whether or not this happens, bringing Schemachine and VLISP's FSBCM closer together brings a kind of closure to our branch of the larger study. Task F (*p.* 14) is one approach to promoting this research to a broader audience.

References cited

[1] W. R. Bevier, W. A. Hunt, J S. Moore, and W. D. Young. An approach to systems verification. *Journal of Automated Reasoning*, November 1989.

[2] Bhaskar Bose. *DDD-FM9001: Derivation of a Verified Microprocessor*. PhD thesis, Computer Science Department, Indiana University, USA, 1994. Technical Report No. 456, 155 pages.

[3] Bhaskar Bose and Steven D. Johnson. DDD-FM9001: Derivation of a verified microprocessor. an exercise in integrating verification with formal derivation. In G. Milne and L. Pierre, editors, *IFIP Conference on Correct Hardware Design and Verification Methods*, pages 191–202. Springer, LNCS 683, 1993. also published as Technical Report 380, Computer Science Department, Indiana University.

[4] Bhaskar Bose, M. Esen Tuna, and Steven D. Johnson. System factorization in codesign: A case study of the use of formal techniques to achieve hardware-software decomposition. In *International Conference on Computer Design*, pages 458–461. IEEE, October 1993. Also published as Tech Report No. 386, Computer Science Department, Indiana University.

[5] C.D. Boyer and Steven D. Johnson. Using the digital design derivation system: Case study of a VLSI garbage collector. In Darringer and Ramming, editors, *Ninth International Symposium on Computer Hardware Description Languages*, Amsterdam, 1989. IFIP WG 10.2, Elsevier. Also published as Technical Report 274, Computer Science Dept. Indiana University, April 1989.

[6] Bishop C. Brock, Jr. Warren A. Hunt, and Matt Kaufmann. The fm9001 microprocessor proof. Technical Report 86, Computational Logic, Inc., `http://www.cli.com/`, December 1994. 1410 pages.

[7] Robert G. Burger. The scheme machine. Technical Report 413, Indiana University, Computer Science Department, August 1994. 59 pages.

[8] William Clinger and Jonathan Rees. Revised report on the algorithmic language scheme. Technical Report 341 (rev. 4), Indiana University, November 1991.

[9] William C. Clinger. The Scheme 311 compiler: an exercise in denotational semantics. In *Proc. Symposium on Lisp and Functional Programming*, pages 356–364. ACM, 1984.

[10] William C Clinger. The scheme 312 version 4 reference manual, 1985. Unpublished supplementary course material.

[11] David L. Dill, Seungjoon Park, and Andreas G. Nowatzyk. Formal specification of abstract memory models. In Gaetano Borriello and Carl Ebeling, editors, *Research on Integrated Systems: Proceedings of the 1993 Symposium*, pages 38–52. The MIT Press, 1993.

[12] Arthur D. Flatau. A verified implementation of an applicative language with dynamic storage allocation. Technical Report 83, Computational Logic, Inc., `http://www.cli.com/`, January 1993.

[13] Anthony C. J. Fox and Neal A. Harman. An algebraic model of correctness for superscaler microprocessors. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design*, pages 346–361, Berlin, 1996. Springer. LNCS No. 1166.

[14] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. McGraw Hill, 1992.

[15] Donald I. Good, Matt Kaufmann, and J Strother Moore. The role of automated reasoning in integrated system verification environments. Technical Report 73, Computational Logic, Inc., January 1992. URL: http://www.cli.com.

[16] Brian T. Graham. *The SECD Microprocessor: A verification case study*. Kluwer, 1992.

[17] Joshua Guttman, John Ramsdell, and Mitchell Wand. Vlisp: A verified implementation of scheme. *Lisp and Symbolic Computation*, 8(1/2):5–32, 1995. Special issue on VLISP.

[18] Steven D. Johnson. Applicative programming and digital design. In *Proc. Eleventh Annual ACM SIGACT-SIGPLAN , Symposium on Principles of Programming (POPL'84)*, pages 218–227, 1984.

[19] Steven D. Johnson. *Synthesis of Digital Designs from Recursion Equations*. MIT Press, Cambridge, 1984. ACM Distinguished Dissertation 1984.

[20] Steven D. Johnson. Digital design in a functional calculus. In Milne and Subramanyam, editors, *Formal Aspects of VLSI Design*, pages 153–178. North-Holland, Amsterdam, 1986. Proceedings of the 1985 Edinburgh Workshop on VLSI.

[21] Steven D. Johnson, B. Bose, and C.D. Boyer. A tactical framework for digital design. In Birtwistle and Subramanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 349–383. Kluwer, Boston, 1988.

[22] Steven D. Johnson, R.M. Wehrmeister, and B. Bose. On the interplay of synthesis and verification: Experiments with the FM8501 processor description. In Claesen, editor, *Applied Formal Methods for Correct VLSI Design*, pages 385–404. Elsevier, 1989. IMEC 1989.

[23] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996. The Java(tm) Series.

[24] Paul S. Miner and Steven D. Johnson. Verification of an optimized fault-tolerant clock synchronization circuit: A case study exploring the boundary between formal reasoning systems. In Satnam Singh, Mary Sheeran, and Geraint Jones, editors, *Designing Correct Circuits*, Springer. Springer, 1996. Electronic Workshops in Computing, `http://www.springer.co.uk/ewic/workshops/DCC96`.

[25] Paul S. Miner, Shyamsundar Pullela, and Steven D. Johnson. Interaction of formal design systems in the development of a fault-tolerant clock synchronization circuit. In *13th Symposium on Reliable Distributed Systems*, pages 128–137, 1994. Held at Dana Point, California, October 1994.

[26] John D. Ramsdell. Personal communication.

[27] Kamlesh Rath, Bhaskar Bose, and Steven D. Johnson. Derivation of a DRAM memory interface by sequential decomposition. In *Proceedings of the International Conference on Computer Design*, pages 438–441. IEEE, October 1993. Also published as Tech Report No. 385, Computer Science , Department, Indiana University.

[28] V. Swarup, W. M. Farmer, J. D. Guttman, L. G. Monk, and J. D. Ramsdell. The VLISP byte-code interpreter. M 92B097, The MITRE Corporation, September 1992.

[29] Vipin Swarup, William M. Farmer, Joshua D. Guttman, Leonard G. Monk, and John D. Ramsdell. VLISP byte code interpreter. Techni-

cal Report M92B097, The MITRE Corporation, September 1992. Available through the the Scheme repository, `ftp://ftp.cs.indiana.edu/pub/-scheme-repository/doc/pubs/vlisp/`.

[30] Mitchell Wand. Deriving target code as a representation of continuation semantics. *ACM Transactions on Programming Languages and Systems*, 4(3):496–517, July 1982.

[31] Mitchell Wand. Semantics-directed machine architecture. In *Conf. Rec. 1st ACM Symposium on Principles of Programming Languages*, pages 234–241, 1982.

[32] R.M. Wehrmeister. Derivation of an SECD machine: Experience with a transformational approach to synthesis. Technical Report 290, Indiana University, Computer Science Department, September 1989.