INDIANA UNIVERSITY
COMPUTER SCIENCE DEPARTMENT

TECHNICAL REPORT NO. 409

Continuations in Hardware-Software Codesign

M. Esen Tuna, Steven D. Johnson, and Robert G. Burger

JULY 1994

To appear in the proceedings of the *1994 IEEE International Conference on Computer Design* (*ICCD '94*), Cambridge, Massachusetts, October, 1994.

Continuations in Hardware-Software Codesign *

M. Esen Tuna Steven D. Johnson Robert G. Burger

Indiana University Computer Science Department
Bloomington, Indiana
{mtuna,sjohnson,burger}@cs.indiana.edu

Abstract

This paper presents a case study for using high-level programming techniques to support the migration of software into hardware. The example is a derived implementation of a symbolic processing machine. The design environment employs codesign to maintain consistency between an executable software model of the system and the individual hardware components that are extracted from it. The presentation focuses on the use of continuations to move from a procedural view of memory allocation to a process view. Our previous work has used functional models as a source for correct hardware derivation using a transformational algebra. The work reported here will result in extensions that deal more powerfully with the factorization of sequential subsystems.

KEYWORDS AND PHRASES: hardware-software *codesign*, hardware description languages, formal methods for hardware, hardware emulation, transformational programming, design derivation, Scheme.

1 Introduction

This paper illustrates how a set of high-level programming techniques is used in a system prototyping environment to incrementally transfer executable design specifications into hardware. In this specialized context, codesign issues arise because we want to maintain the executability of the design model as its components are realized in hardware one by one. It is not clear whether the particular set of techniques described here would be applicable in production codesign situations. However, we believe that this work illustrates an important class of language features to consider for a successful codesign environment. It is certainly the case that for the restricted codesign applications to which they have been applied, these techniques have provided leverage over *software map-over* problems.

The environment we have developed is based on the programming language Scheme [1], a Lisp dialect which serves as the framework implementation language, as a formal modeling language for system specification, and with certain restrictions as a hardware description language [2, 3]. We exploit some advanced features of Scheme—most notably its support of functional values with appropriate scoping rules—in the techniques we describe below. These techniques are standard in functional programming methodology, but this is the first time they have been applied directly to hardware descriptions. Of course, analogous techniques could be developed in any modeling language, but this particular approach requires a properly scoped, higher-order modeling language.

The design environment we have developed allows us to integrate the hardware realization of a system component directly with the original software model. This provides a means to seemlessly stage individual subsystems into hardware while maintaining a coherent, animated, global view of the system.

These techniques have been applied to the design of VLSI emulations and moderate-scale demonstration prototypes. In our experimentation so far, the goal has been a full hardware realization with no software components in the final product. Codesign is involved *during* design in maintaining the global system description during its transition to hardware. In this context, the benefits include greater longevity of functional tests and more realism in design exploration and validation. In addition, we have been able to evaluate partial designs earlier and more cheaply, since we can incorporate and exercise hardware components individually. Similar high-level language techniques should provide comparable benefits in other codesign

*Research reported herein was supported, in part, by NSF: The National Science Foundation, under grants numbered MIP 89-21842 and MIP 92-08745, and a Graduate Research Fellowship.

settings, especially where experimentation is needed to explore the hardware-software boundary.

1.1 The Codesign Exercise

The codesign exercise is a specialized processor and memory system for execution of compiled Scheme. This exercise is part of a research project to extend a software-oriented methodology [4] to include hardware implementation targets.

We begin with a very-high-level specification, a simple Scheme interpreter representing a user-level definition of the language. Using existing techniques of the software methodology, the naive language specification is factored into a recursive compiler and an iterative machine model.

The initial specification is abstract in that it allows the modeling language to provide the storage object for the modeled machine. As Figure 1 illustrates, the

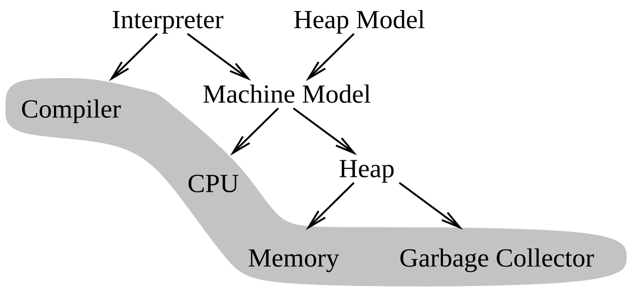


Figure 1: Road map for a μ-processor derivation

machine is further decomposed to incorporate a heap-storage component. The heap object is self-managing; it contains its own allocator, storage recycler, and other mechanisms that support symbolic processing. The shaded area shows the final components of the project.

Figure 2 shows the resulting architecture of the design, which consists of four processors sharing a dual-ported memory. There are four possible implementation configurations of the system when three of the processors along with the primitive memory are grouped as the heap object. Both the CPU and the heap may be implemented in either hardware or software. Initially, the entire system is modeled at a behavioral level in software. Using tools that we have developed [3], the executable subsystem descriptions are individually transformed into hardware. Figure 3 shows a prototype in which the heap system and the CPU have been realized in this way. The prototype environment, shown in Figure 3, consists of an uncommitted wire-wrap area, banks of lights and

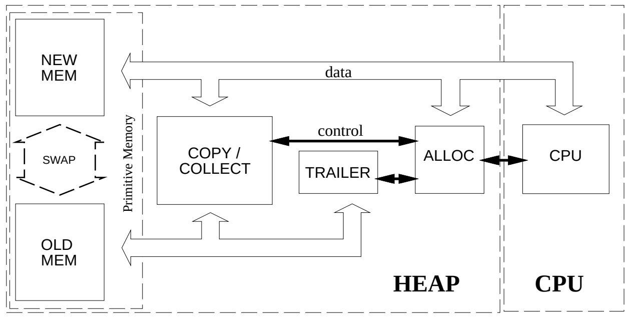


Figure 2: System architecture of the Scheme Machine

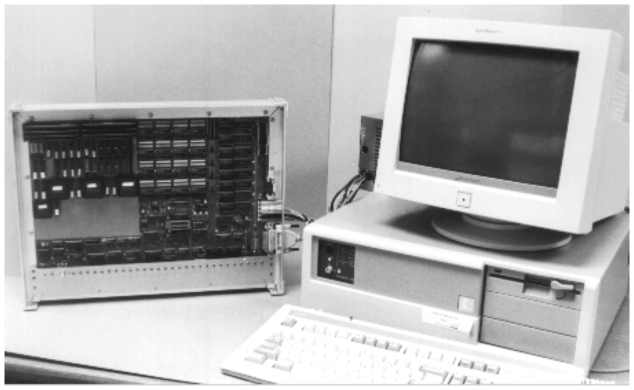


Figure 3: Scheme Machine with the software host PC

switches, and other facilities unused in this project [5, 6]. The computer to the right is connected to the project board and is used for simulating the software components, testing the hardware components, and evaluating the entire system.

Typically a design starts with an all-software model, and the modules of the design are gradually mapped onto hardware. It is crucial that the system be testable at each stage in the design process. Hardware and software components co-exist throughout the design, even if the final target is an all-hardware implementation. We start with software models—executable in Scheme—of both the CPU and the heap.

Next the heap is implemented in hardware. At this stage the behavior of the hardware heap is tested against its software specification, which is also an executable model. The computer, running Scheme, executes the specification of the CPU model against the heap implementation. Since the hardware heap is an exact implementation of the software model, the software CPU model can use either the software heap model or the hardware heap through interface routines and modules. Finally, the CPU model is implemented in hardware running off the hardware heap, completing the hardware implementation of the system.

2 Scheme, Continuations, and Behavioral Description

Scheme is a lexically scoped dialect of Lisp, an expression-oriented symbolic processing language. Its support of functional (procedural) objects having full status as data values is central to the techniques we develop later. Space permits only a brief sketch of this feature; tutorial material can be found in [7, 4].

The Scheme expression `(lambda (`$x_1 \cdots x_n$`)` e`)` returns a function with defining expression e involving parameters $x_1, \ldots, x_n$. A `lambda` expression may occur anywhere that an expression is permitted; functions may be created, passed as arguments, stored in structures, and of course, applied to arguments. The most common use of function expressions is to define procedures at top level, but function expressions are also used to represent data structures and objects.

The techniques of this paper involve a more subtle use of functional values to specify control flow. A *continuation* represents the future course of a computation, mapping the result of past execution to the final outcome of the program. As a first illustration, consider the system below, which defines a version of the *factorial* function, `FAC`, together with a multiplication algorithm, `MPY`:

```
(define FAC1
   (lambda (n m)
     (if (= n 0)  m
         (FAC1 (- n 1) (MPY1 n m)))))
(define MPY1
   (lambda (n m)
     (if (= n 1)  m
         (+ m (MPY1 (- n 1) m)))))
```

In the following version of `MPY`, an explicit continuation replaces the implicit recursion of the original version. The parameter `k` tells $\texttt{MPY}_2$ "what to do with the result." The initial continuation is to resume the calculation of `FAC`. Within the body of `MPY`, the continuation records what additions to do in the future. An effect of introducing the continuation is that it makes the system of definitions more iterative by moving calls to `FAC` and `MPY` into tail-recursive positions.

```
(define FAC2
  (lambda (n m)
     (if (= n 0)  m
         (MPY2 n m (lambda (x)
                     (FAC2 (- n 1) x))))))
(define MPY2
  (lambda (n m k)
     (if (= n 1)
         (k m)
         (MPY2 (- n 1) m (lambda (x)
                           (k (+ m x)))))))
```

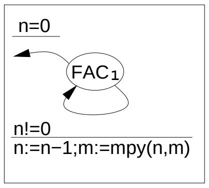
Further refinements to the system are based, in part, on analysis of the continuations:

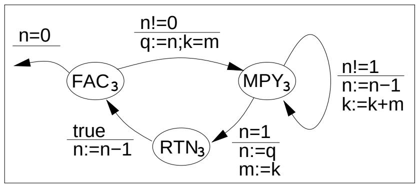
(a) `(lambda (x) (FAC`$_2$` (- n 1) x))` captures the value of `n` for use later. In the version below, this value is passed through variable `q` in `MPY`. Since there are no other values to capture within the scope of `FAC`, the continuation can be defined externally as the function `RTN`.

(b) `(lambda (x) (k (+ m x)))` simply adds `m`'s to the result, and can be therefore represented by an accumulator, `k`.

```
(define FAC3
  (lambda (n m)
     (if (= n 0)
         m
         (MPY3 n m n m))))
(define MPY3
  (lambda (n m q k)
     (if (= n 1)
         (RTN3 q k)
         (MPY3 (- n 1) m q (+ k m)))))
(define RTN3
  (lambda (n m)
     (FAC3 (- n 1) m)))
```

We use systems of tail-recursive function definitions, like the one above, as a behavioral hardware description language [2, 3]. The functions represent states of an extended finite-state machine, which in this example would look like:





Before leaving this example, let us briefly review it from the perspective of formal verification. The first transformations that introduce continuations are fairly standard. Although we have not yet mechanized them, it would be reasonable to do so. Getting from the second to the third version of `FAC` was a more subtle undertaking. These refinements are less likely to be automated, and in the case that they are not, another reasoning system might be needed to verify them [8].

Although formal verification is the underlying motive of this research, this article also has to do with practical techniques for getting from the specification (or model) of an object to its realization in hardware.

3 Using Continuations in Factorization

Our notion of factorization is a general form of abstraction where parts of a design are encapsulated as a named entity. The factored component may be a complex valued stream (signal) in the circuit or a set of operations on the signals [9]. Factorization may involve a sequential or combinational component. We also view sequential state machine decomposition as factoring a set of states from the original design and creating a coprocess which is communicating with the residual system.

Maintaining the integrity and executability of the model is a central issue in codesign. In this section we introduce the use of continuations to provide for process creation and decomposition without losing model continuity [10]. Continuations provide a smooth and seamless transition from subroutines to processes and further to simple coprocesses.

3.1 Scheme Machine Exercise

We consider the execution of a *cons* operation on an abstract Scheme machine. *Cons* is a primitive in Scheme to construct lists. To execute *cons*, first the arguments are computed, and then a memory object called a *consbox* is allocated. The *consbox* has two fields into which the arguments are written. $\texttt{Machine}_0$ in Figure 4 is a fragment of a state machine representing the Scheme Machine model. We consider some

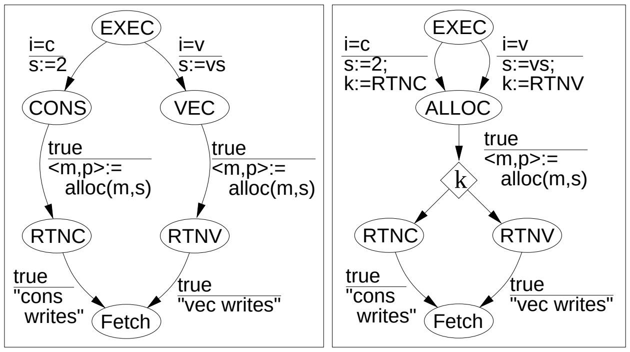


Figure 4: $\texttt{machine}_0$ and $\texttt{machine}_1$

of the states involving memory allocation to illustrate the use of continuations in factorization. The EXEC state computes the arguments of the *cons* operation, followed by the CONS state, where a *consbox* is allocated. The memory allocation is done by the function alloc, which given the old memory, m, and the size, s, of the desired allocation, returns the new memory configuration and a pointer, p, to the allocated *consbox*. Finally RTNC state writes the two arguments into the *consbox* and calls the Fetch state.

In the initial specification of the Scheme Machine ($\texttt{machine}_0$), the memory allocator (alloc) is a proper subroutine in the behavioral model rather than a separate state machine. Our goal is to transform this initial model of the machine into a target model where the machine and allocator are interactive processes. To facilitate this decomposition we first incorporate the allocation as a new state, ALLOC, into the machine. In Figure 4 alloc is invoked during transitions from CONS, whose continuation is RTNC, and VEC, whose continuation is RTNV. In $\texttt{machine}_1$ we introduce a new state, ALLOC, as a common control path for the two allocations. We now need a mechanism to proceed with either continuation. For the moment we can simply pass the continuation itself, RTNC or RTNV, through a new register k. We depict the conditional state transitions following ALLOC as transitions from ◇k, representing an explicit assignment into the state register of the machine. This transformation results in a net reduction of one state at this level of abstraction. However, the real reduction is much greater, since ALLOC will next be replaced by a sequential protocol.

Returning to the *factorial* example in Section 2, we could save the continuation at the state $\texttt{FAC}_3$ by adding another register, r, to hold the state name $\texttt{RTN}_3$. Now MPY simply applies r to invoke the continuation.

```
(define FAC4
  (lambda (n m)
     (if (= n 0)
         m
         (MPY4 n m n m RTN4))))
(define MPY4
  (lambda (n m q k r)
     (if (= n 1)
         (r q k)
         (MPY4 (- n 1) m q (+ k m) r))))
(define RTN4
  (lambda (n m)
     (FAC4 (- n 1) m)))
```

The assumption in $\texttt{machine}_1$ (Fig. 4) is that the result of the allocation is immediately available. However, we would like to factor out the allocation from $\texttt{machine}_1$ in order to make allocation a concurrent subprocess communicating with the machine. As shown in Figure 5, we split the state ALLOC, keeping the the continuation $\texttt{machine}_2$ and creating a new state machine (allocator) which also has a state called alloc generating the result of the allocation. We also generate handshake signals, req and done,

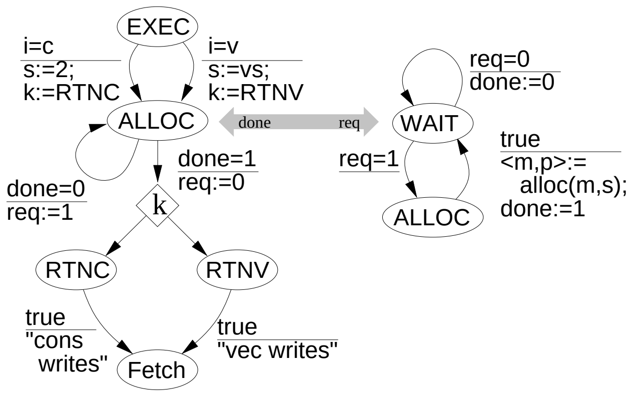


Figure 5: $\texttt{machine}_2$ with `allocator state machine`

between the two state machines and introduce wait loops as part of the communication protocol. In this example we have introduced a protocol in an *ad hoc* manner; however, our research addresses sequential decomposition formally as discussed in [11].

In the `ALLOC` state, $\texttt{machine}_2$ sets the `req` signal *high* and waits in an idle loop until the `done` signal is set by the `allocator`. Then the new memory and the pointer become available, and execution proceeds with the continuation saved earlier. The `allocator` is a coprocess waiting in a tight idle loop until it receives a `request`, upon which it enters its `ALLOC` state and signals `done` upon completion of the allocation. Now the serial operations (like garbage collection) of the `alloc` function can further be assigned to new states of the `allocator` without the possible state explosion problem in the initial specification, if `alloc` would be serialized in place.

4 Concluding Remarks

4.1 Evolution of the Project

Programming language research developed various techniques to construct compilers and machine specifications from fully abstract language descriptions. The Scheme Machine project is an extension of that research into hardware. We started with a complete executable behavioral specification which allows us to validate, animate, and explore the system design at an abstract level. In the initial specification of the system, the subsystems are modeled as subprocedures, avoiding a premature partitioning of the system, which is a key problem in codesign[12].

We then elected an implementation of a primitive memory—a dual ported DRAM capable in certain instances of performing two operations in 125% of a cycle. The memory unit also deals with refresh internally using time–division multiplexing on the memory bus. Now the memory has become a process rather than a subroutine. However, this change is transparent to the rest of the system, since the memory continues to appear as a vector in the Scheme model.

Later on, the garbage collector and the trailer process were staged onto the board. The collector is tested with heap images downloaded in the memory. At this point the CPU was redesigned, and we have reconsidered the codesign issues involved to interface the hardware and software components of the project. This paper describes the process of gradually mapping an initial functional specification onto hardware while maintaining the integrity of the system with its software and hardware components.

4.2 Technology Transfer to Practical CAD

The work presented in the paper comes from a context in which design techniques—and the tools that derive from them—relate to a particular programming methodology. The Scheme language has evolved hand-in-hand with this methodology. There is no accessible counterpart to "continuation" in conventional languages (although C's `setjmp` is a related construct) and certainly not in existing hardware description languages. One can say, on the one hand, that continuations arose as a mechanism to selectively impose control qualities on applicative specifications that are inherent to imperative languages. The idea is to gain control over *control*, just as in most languages one has control over data representations.

A central problem in codesign is to bridge the software-hardware "abstraction gap" [13]. This notion presumes that we are able to specify at a high enough level to obscure the boundaries of hardware and software, or at least the interface between those two. Equivalently, we must obscure the distinction between *procedure* and *process*, and this entails manipulation of control. Thus, one would suspect that constructs like continuations would be involved in the abstraction.

On the other hand, there clearly are ways to handle the examples presented in this paper in current HDLs. As the state diagrams show, the end product introduces a variable to encode the continuation. One could simply save this value in a register and then branch according to it later. In behavioral HDLs with a fixed *next-state* register one could assign a stored value to it (we know of no HDL that allows this, though).

The techniques presented here are advantageous in two ways. First, they allow us to defer representation decisions for control; we were able to incrementally embellish the behavioral specification with more control states. Second, the refinements had minimal impact on the behavioral model and could be readily dealt with in the prototyping activity.

We believe that the goals and problems in codesign are going to necessitate greater flexibility at the specification level than currently exists in HDLs. There needs to be more interaction between software and hardware subsystem development [10]. There needs to be less bias toward software or hardware in modeling languages and their affiliated tools [12]. We found it encouraging that control manipulation techniques that have been developed in software-oriented research were so readily adapted to hardware specification problems and prototyping activities.

5 Acknowledgements

The research described here is part of a large project with a substantial context. Of the numerous individuals involved in this project, several have made direct contributions to this paper. We have had many fruitful discussions with Kamlesh Rath, who also was involved with implementing the garbage collection subsystem described above. Willie Hunt has helped develop both the hardware and the software of the codesign environment, and has contributed critical engineering expertise to the project. Hunt designed and built the primitive memory subsystem. Peter Weingartner participated in early experimentation with continuations. Last, but by no means least, David Boyer has been involved with several stages of the computer derivation project, including an earlier derivation of the CPU and garbage collector, and implementation of the software environment for the study.

References

[1] W. Clinger and J. Rees, "The revised^4 report on the algorithmic language Scheme," *Lisp Pointers*, vol. 4, no. 3, pp. pages 1–55, 1991.

[2] S. D. Johnson and B. Bose, "A system for mechanized digital design derivation," in *Proceedings of ACM International Workshop on Formal Methods in VLSI Design* (Subramanyam, ed.), January 1991.

[3] B. Bose, "DDD - A Transformation system for Digital Design Derivation," Tech. Rep. 331, Indiana University, Computer Science Department, May 1991.

[4] D. P. Friedman, M. Wand, and C. T. Haynes, *Essentials of Programming Languages.* The Massachusetts Institute of Technology, 1992.

[5] D. Winkel, F. Prosser, R. Wehrmeister, W. C. Hunt, and C. Hess, "A student VLSI hardware tester," in *Proceedings of the Microelectronic Systems Education Conference and Exposition*, pp. 15–24, 1990.

[6] R. M. Wehrmeister, *Logic Engine User's Manual.* Computer Science Dept. Indiana Univ., Mar. 1991.

[7] G. Springer and D. P. Friedman, *Scheme and the Art of Programming.* McGraw-Hill, 1990.

[8] B. Bose, S. D. Johnson, and S. Pullela, "Integrating boolean verification with formal derivation," in *Proceedings of IFIP Conference on Hardware Description Languages and their Applications* (D. Agnew, L. Claesen, and R. Camposano, eds.), pp. 127–134, Elsevier, Apr. 1993.

[9] B. Bose, M. E. Tuna, and S. D. Johnson, "System factorization in codesign: A case study of the use of formal techniques to achieve hardware-software decomposition," in *Proceedings of the International Conference on Computer Design*, pp. 458–461, IEEE, Oct. 1993.

[10] S. Kumar, J. H. Aylor, B. W. Johnson, and W. A. Wulf, "A framework for hardware/software codesign," *IEEE Computer*, pp. 39–45, Dec. 1993.

[11] K. Rath, B. Bose, and S. D. Johnson, "Derivation of a DRAM memory interface by sequential decomposition," in *Proceedings of the International Conference on Computer Design (ICCD)*, pp. 438–441, IEEE, Oct. 1993.

[12] N. S. Woo, A. E. Dunlop, and W. Wolf, "Codesign from cospecification," *IEEE Computer*, pp. 42–47, Jan. 1994.

[13] K. ten Hagen and H. Meyr, "Partioning and surmounting the software-hardware abstraction gap in an asic design project," in *IEEE International Conference on Computer Design*, pp. 462–465, Oct. 1993.

[14] K. Rath, M. E. Tuna, and S. D. Johnson, "Specification and synthesis of bounded indirection," Tech. Rep. 398, Indiana University, Computer Science Department, Feb. 1994.

[15] C. Boyer and S. D. Johnson, "Using the digital design derivation system: Case study of a VLSI garbage collector," in *Ninth International Symposium on Computer Hardware Description Languages* (Darringer and Ramming, eds.), (Amsterdam), IFIP WG 10.2, Elsevier, 1989.

[16] S. D. Johnson, B. Bose, and C. Boyer, "A tactical framework for digital design," in *VLSI Specification, Verification and Synthesis* (Birtwistle and Subramanyam, eds.), pp. 349–383, Boston: Kluwer, 1988.

[17] P. A. Subrahmanyam, G. D. Micheli, and K. Buchenrieder, "Hardware-software codesign," *IEEE Computer*, pp. 84–87, Jan. 1993.