DRAFT MATERIAL

August 4, 1994

Appendix material for "Studies of the Single-pulser in Various Reasoning Systems," by Steven D. Johnson, Paul S. Miner, and Shyam Pullela.

DDD derivation of the single pulser

The file called `sp.ss`, shown in Figure , contains three Scheme function definitions. The first of these, `SP`, is a representation of the Single-pulser state machine with states `READY` and `WAIT`. The `let`-bindings `X` and `I` illustrate one way that DDD specifications are interfaced to the Scheme I/O system for execution of the specification. In the body of `SP`, the binary `IN`put value is coerced to a truth value, `I`. This is an expedient way to deal with the coercion from binary to truth-value, but in other circumstances might not be appropriate for modeling.

Again in `SP`, the preamble

```
(lambda (O)
  (let ((X (out O))
        (I (= (inp IN) 1)))...
```

Could have been suppressed by a syntax extension to give a more succinct way of writing down the state machine. It is omitted here because, even with the extended syntax, the user must still understand how DDD is operating on the more primitive syntax.

Figure shows an execution of `SP`.

The derivation

As we will see, much of the derivation has to do with isolating the I/O interface from the part of the specification actually intended for hardware. The advantage of including the I/O is that each transformation yields a program with which we can interact. Thus, it is very common to see I/O modeling in DDD specifications. This section contains an annotated copy of a derivation script. In practice, the user would develop this script interactively, stopping occasionally to consolidate the relevant commands as a Scheme source file.

Figure is a chart of the derivation that we are about to explain. The first page of the figure, Fig. (a) shows those steps that isolate the circuit description. After this phase, the guided work of the dervivation is essentially finished. The second page shows the tedious but entirely routine process of reducing the design into a form that can be project to layout by logic synthesis tools. This second phase consists of syntax transduction, instantiation of constants, and logic optimization.

Each rectangle in the figure represents set of Scheme `define` expression. The names appearing inside the rectangls are those which are thereby defined. The figure also shows that many of the boxes have names, which are shown as attached to the boxes. A dotted boundary indicates a data structure that is either implicitly built, for example, by a global definition.

We now enter DDD and read in the file `sp.ss`

```
;; Load DDD system & partitioning software
;; --------------------------------------
(load "/u/bose/DDD/V1/ddd.so")
(load "/u/psm/ddd/sync/patch.ss")

;; Initial specification and sequential system
;; ------------------------------------------
(define spbeh     (read-file "sp.ss"))
(define SEQSYS    (itrsys->seqsys spbeh))
(define SELECT    (seqsys.select SEQSYS))
(define SYSTEM    (seqsys.system SEQSYS))
(define BASE      (seqsys.base   SEQSYS))
```

Next, a construction is performed translating `SP` into an equivalent network of streams. This step represents the transition between a behavioral mode of description and a structural mode. Construction `itrsys->seqsys` produces a triple,

$$\texttt{SEQSYS} = \texttt{(SELECT SYSTEM BASE)}$$

as shown in Figure . The decision combinator `select` represents the interface between a control automaton and the data path of the resulting sequential system; it abstracts the control structure of `spbeh`. The `lambda*` in `select`'s definition is not the standard Scheme `lambda`; it permits the nested formal parameter structure seen in the definition. This definition gives the discrete version of `select`, which operates on individual values. The function `SELECT`, defined in a moment, is the extension of `select` to streams.

The function `SP` in Figure is the system itself, and it consists mainly of stream definitions representing traces of the individual variables `x`, `i`, and `o` in `spbeh`. In addition, there is a signal `STATE` implementing the control automaton, and a group of predicate signals called `STATUS` which feed back instances of `SELECT`.

The function `XPS` in the defining expression of `STATUS` coerces the pair of streams, `(STATE I)` into a stream of pairs:

$$\texttt{((STATE}_0\ \texttt{I}_0\texttt{)}\quad \texttt{(STATE}_1\ \texttt{I}_1\texttt{)}\quad \texttt{(STATE}_2\ \texttt{I}_2\texttt{)}\quad \ldots\texttt{)}$$

This is a very commonplace coercion in DDD expressions. `XPS` is actually a syntax extension, a generic function that, in a more strongly typed system, would have to be instantiated for each n-tuple.

The third component of SEQSYS is a collection of base definitions generated by the itrsys->seqsys construction. As mentioned earlier, SELECT is the extension of select to streams; two symbolic tokens, ready and wait, are needed to implement the STATE machine; finally, READY and WAIT are the stream versions of these new tokens.

In the next derivation step, the goal is to isolate the modeling interface and at the same time simplify the SELECT expressions:

```
(define SYSTEM.1
  (let* ((SYSTEM.1a (expand-stream 'O SYSTEM))
         (SYSTEM.1b (identify-stream 'O
                      '(SELECT STATUS ZERO ZERO ONE) SYSTEM.1a))
         ;; The previous two steps isolate the O register and move it into
         ;; the modelling interface
         (SYSTEM.1c (remove-useless-selects SYSTEM.1b))
         ;; Flatten nested selects
         )
    SYSTEM.1c))
```

The let* is a standard Scheme expression for cascaded let-binding. In the first step, SYSTEM.1a all occurances of O are expanded according to O's defining expression in system SP (Figure). Next all occurances of (SELECT STATUS ZERO ZERO ONE) are identified as O. The affect is that subexpressions (OUT O) have been replaced by (OUT (! zero O)), which will later be factored out of the design as part of the modeling interface. The transformation remove-useless-selects eliminates applications of SELECT in which all the alternatives are indentical. The resulting system, called SYSTEM.1 is shown in Figure .

Why is there an XPS *in* SYSTEM.1*?*

The next step of the derivation is to segregate that portion of the design that will be realized in hardware. As the commentary states, two derivation paths were explored at this point. We shall follow the first all the way to hardware.

```
;; Now we have two alternatives to partition the system
;; 1) First, partition out the interface (X,I) and then the STATE
;; 2) First, partition out the interface along with the STATE
;;    (X,I,STATE) and then partition STATE from the resulting system.

;; Alternative #1
;; --------------
(define PARTITION1 (partition '(X I) (list SELECT SYSTEM.1 BASE)))
(define PARTITION1.1 (partition '(STATE) (partition.1 PARTITION1)))

(define SYSTEM.2 (seqsys.system (partition.1 PARTITION1.1)))
(define SELECT.2 (seqsys.select (partition.1 PARTITION1.1)))

(define SEQSYS.3 (optimize-seqsys SELECT.2 SYSTEM.2))
```

```
(define SYSTEM.3 (seqsys.system SEQSYS.3))
(define SELECT.3 (seqsys.select SEQSYS.3))

(define STATE-STREAM (extract-stream 'STATE
                       (seqsys.system (partition.2 PARTITION1.1))))
;; ------------------
;; End Alternative #1
```

The `partition` construction decomposes one system object—recall that a system object is a triple consisting of a decision combination, a sequential system description, and a collection of base definitions—into two subsystems. We construct `PARTITION1` by extracting streams `X` and `I` from `SYSTEM.1` (Figure . The resulting subsystems are shown in Figure . The subsystem `SP.1` in the figure contains that portion of the design intended for hardware. The residual system, `SP.2` contains those constructs that derive from the modeling interface. Although `select` is a component of the second subsystem it is not used in the system `SP.2`.

`SP.1` is further partitioned in order to isolate the `STATE` signal. The result is shown in Figure . `SYSTEM.2`, `SELECT.2`, and later, `SYSTEM.3` and SELECT.3", are simply fields of subsystem objects. `SEQSYS.3` is a refinement of `SEQSYS.2` in which applied instances of the `select` combination are simplified. In this case, partial evaluation of the expression `(SELECT STATUS ZERO ZERO ONE)` reveals that this instance of the underlying `select` can be reduced. The result is shown in Figure Extraction of the `STATE-STREAM` signal marks the beginning step of projecting the design into a realizable boolean system.

```
;; Now derive nextstate combinator
;; -------------------------------
(define NEXT-STATE
  (optimize-sel
   (partial-eval STATE-STREAM SELECT)))

;; Load the representation mappings
;; --------------------------------
(load "rep.ss")

;; Map nextstate to binary representation.
;; ---------------------------------------
(define NEXTSTATE.BIN.1
  (rename-sel 'next-state 'K NEXT-STATE))
(define NEXTSTATE.BIN.2
  (project-sel state.map NEXTSTATE.BIN.1))
(define NEXTSTATE.BIN
  (map optimize-sel
       NEXTSTATE.BIN.2))
```

The file rep.ss, is shown in Figure contains the declarative part of the design specification. The unusual form of the definitions is an artifact of a project to compute type translations by a nonstandard interpretation of the source expression. A switchover to a more conventional syntax—probably that of the Infer type system—is planned. The NEXT-STATE definition, at the beginning of Figure , is obtained by symbolically expanding SELECT according to its definition. After a sequence of four elementary steps, we get a simplifiable defining expression for the STATE signal:

Why/how is READY *equated with* STATE *here?*

How can I *be free in the definition of* STATE*?*

```
(case s
  [ready (if I 1 0)]
  [wait  (if I 1 0)]))
```

Returning to the derivation script, we make a similar projection of the remaining select expression in SEQSYS.3

```
;; Map select to binary representation.
;; ------------------------------------

(define ENCODE.BIN1 (rename-sel 'select 'CMD SELECT.3))
(define ENCODE.BIN
  (map optimize-sel
       (project-sel CMD.MAP ENCODE.BIN1)))

;; Project the datapath
;; --------------------
(define DATAPATH.BIN
  (sort-streqns
    (project-streqns streqn.map (system.streams SYSTEM.3) 1)))
```

The results of these transformations are shown in Figure **??**. From this point on, the expressions resulting from the derivation are massaged for input into logic-synthesis optimizers and layout generators. The intermediate expressions are shown in Figure . Details about this phase can be found in [**?**].

```
;; Logic Synthesis
;; ---------------
(define OUTPUT.BOOL (streqns->booleqns DATAPATH.BIN 'd))

(define NEXTSTATE.BOOL
  (symbolic->bit
    (map sel->booleqns NEXTSTATE.BIN) STATE.SYMMAP))

(define ENCODE.BOOL
  (symbolic->bit
    (map sel->booleqns ENCODE.BIN) STATE.SYMMAP))
```

```
(define NEXTSTATE.MIN
  (espresso NEXTSTATE.BOOL 'booleqns 'NEXT-STATE))

(define ENCODE.MIN
  (espresso ENCODE.BOOL 'booleqns 'ENCODE))

;; Final output of the system (registers, control and data-path)
;; -------------------------------------------------------------
(define REGISTER NEXTSTATE.MIN)
(define CONTROL  ENCODE.MIN)
(define OUTPUT   (espresso OUTPUT.BOOL 'booleqns 'SP))
```

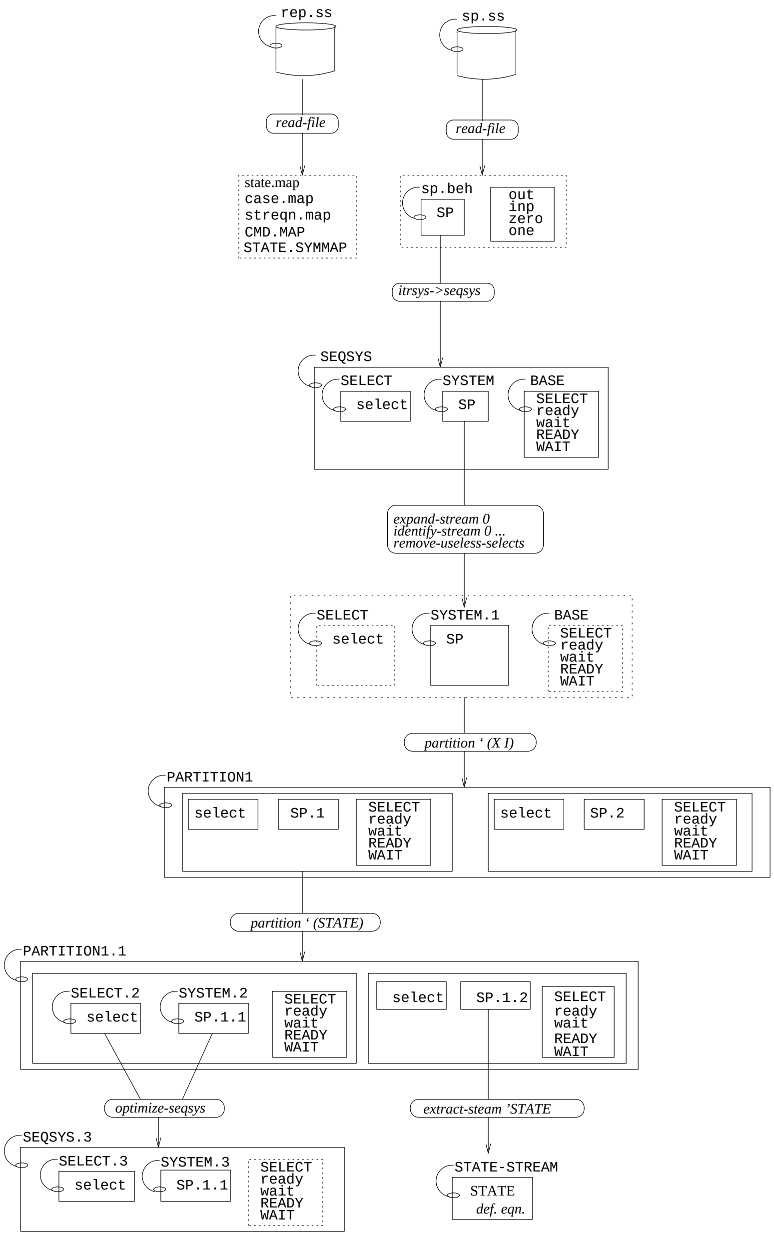
The last three definitions, specifying `STATE`, `CONTROL`, and `OUTPUT` give the reduced implementation of the single pulser, consisting of a single register and a negated-input *and*-gate. In DDD syntax, the final definitions specify the desired network for implementing the single pulser.

```
(define sp
  (lambda (in)
    (letrec
      ((ready (lambda (o)
          (let ((x (out o))
                (i (= (inp in) one)))
             (if i
                 (wait zero)
                 (ready zero)))))

       (wait  (lambda (o)
          (let ((x (out o))
                (i (= (inp in) one)))
             (if i
                 (wait zero)
                 (ready one))))))

      (ready zero))))

(define out
  (lambda (x)
    (display "o=")
    (display x)
    (newline)))

(define inp
  (lambda (i)
    (display "i=")
    (let ([in (read)])
      (if (member in '(0 1))
           in
          (inp i)))))


(define zero 0)
(define one 1)
```

Figure 1: The source file sp.ss

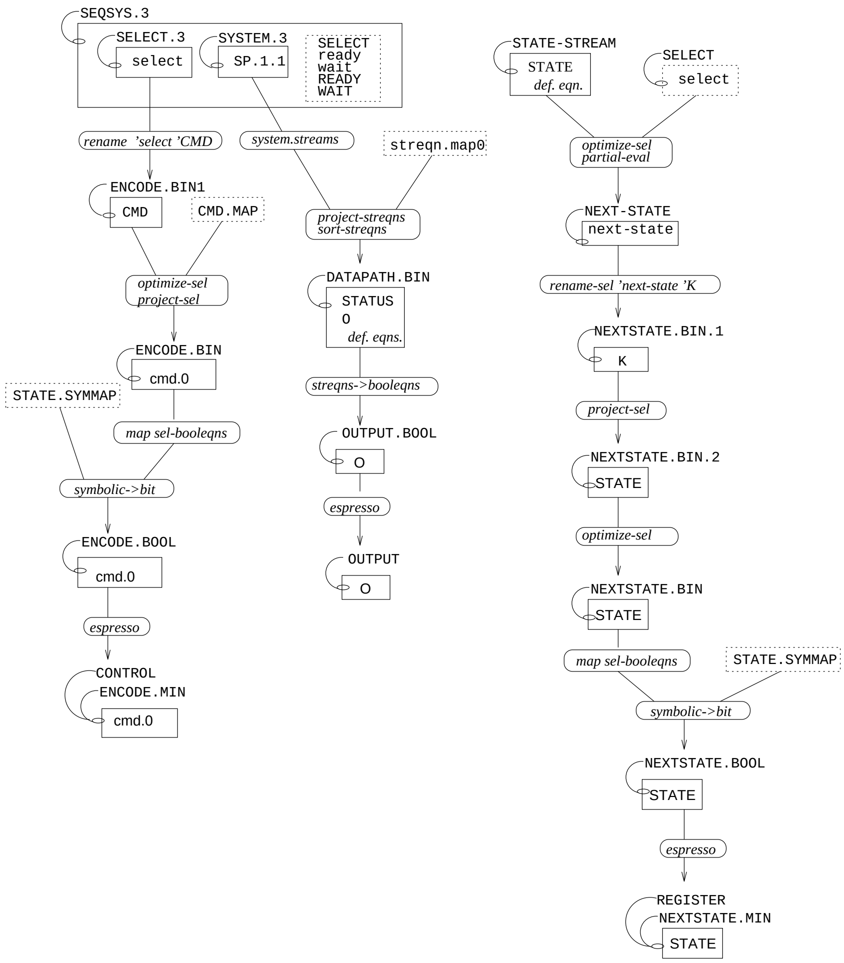
```
Script started on Thu Jun 23 13:17:03 1994
% scheme
Chez Scheme Version 4.1b
Copyright (c) 1991 Cadence Research Systems

> (load "sp.ss")
> (SP ())
o=0
i=1
o=0
i=1
o=0
i=0
o=1
i=0
o=0
i=0
o=0
i=1
o=0
i=0
o=1
i=ĈĈ
debug> a

%
script done on Thu Jun 23 13:18:09 1994
```

Figure 2: Animation of SP



(a)

Figure 3: Chart of the derivation



(b)

Figure 3: Chart of the derivation (continued)

```
(
;;
;; SEQSYS.SELECT
;;
(define select
    (lambda* ((s p0) v0 v1 v2)
       (case s
          [ready (if p0 v0 v1)]
          [wait (if p0 v0 v2)])))
;;
;; SEQSYS.SYSTEM
;;
 (define SP
 (lambda (IN)
 (system-letrec
    ((STATUS (XPS STATE I))
     (I (SELECT STATUS
           (= (INP IN) ONE)
           (= (INP IN) ONE)
           (= (INP IN) ONE)))
     (STATE (! ready (SELECT STATUS WAIT READY READY)))
     (O (! zero (SELECT STATUS ZERO ZERO ONE)))
     (X (SELECT STATUS (OUT O) (OUT O) (OUT O))))
 (list STATUS I STATE O X))))
;;
;; SEQSYS.BASE
;;
 ((define SELECT (select-stream select))
  (define ready 'ready)
  (define wait 'wait)
  (define READY (stream ready))
  (define WAIT (stream wait))))
```

Figure 4: Result of command (itrsys->seqsys spbeh)

```
(define SP
(lambda (IN)
(system-letrec
   ((STATUS (XPS STATE I))
    (I (= (INP IN) ONE))
    (STATE (! ready (SELECT STATUS WAIT READY READY)))
    (X (OUT (! zero O)))
    (O (SELECT STATUS ZERO ZERO ONE)))
   (XPS STATUS I STATE X O))))
```

Figure 5: SYSTEM.1, a refinement to SEQSYS.SYSTEM

```
(
  ;; First subsystem

 ( (define select ...)

   (define SP.1
     (lambda (I)
        (system-letrec
          ((STATUS (XPS STATE I))
           (STATE (! ready (SELECT STATUS
                                   WAIT
                                   READY
                                   READY)))
           (O (SELECT STATUS ZERO ZERO ONE)))
          (list STATUS STATE O))))

   ((define SELECT...) ...)
  )

  ;; Second subsystem

 ( (define select ...)

  (define SP.2
     (lambda (O STATE IN)
        (system-letrec
           ((STATUS (XPS STATE I))
            (I (= (INP IN) ONE))
            (X (OUT (! zero O))))
          (list STATUS I X))))

   ((define SELECT...) ...)
  )
)
```

N.B. The `select` *combinators and base definitions in this partitioning are the same as in Figure and have been manually removed.*

Figure 6: A partitioning applied to `SYSTEM.1`

```
(
 (...

  (define SP.1.1
     (lambda (STATE I)
        (system-letrec
           ((STATUS (XPS STATE I))
            (O (SELECT STATUS ZERO ZERO ONE)))
           (list STATUS O))))

  ...)

 (...

  (define SP.1.2
     (lambda (I)
        (system-letrec
           ((STATUS (XPS STATE I))
            (STATE (! ready (SELECT STATUS
                                    WAIT
                                    READY
                                    READY))))
           (list STATUS STATE))))
  ...)
)
```

N.B. The `select` *combinators and base definitions in this partitioning are the same as in Figure and have been manually removed.*

Figure 7: Further partitioning of `SP.1`

```
(
  ;; Later defined as SELECT.3

  (define select
    (lambda* ((s p0) v0 v1)
       (case s
          [ready v0]
          [wait (if p0 v0 v1)])))

  ;; Later defined as SYSTEM.3

 (define SP.1.1
    (lambda (STATE I)
       (system-letrec
          ((STATUS (XPS STATE I))
           (O (SELECT STATUS ZERO ONE)))
          (list STATUS O))))
)
```

N.B. In this version of DDD, the `optimize-seqsys` *transformation does not produce a triple including base definitions. These would be the same as in Figure .*

Figure 8: Further partitioning of `SP.1`

```
;; Representation file for the singlepulser
;; ---------------------------------------


;; State assignment
;;
(define state.map
  `((K        ,(lambda () '(STATE)))
    (READY    ,(lambda () '(0)))
    (WAIT     ,(lambda () '(1)))
    (ready    ,(lambda () '(0)))
    (wait     ,(lambda () '(1)))
    (p0       ,(lambda () '(I)))
    ))

(define case.map
  `((READY    ,(lambda () '(!STATE)))
    (WAIT     ,(lambda () '(STATE)))
    ))

(define streqn.map
  `(
    ;; Inputs
    (I        ,(lambda () '(I)))
    (STATE    ,(lambda () '(STATE)))
    ;; Stream equations & unique-?&NOP's
    (O        ,(lambda () '(O)))
    (ZERO     ,(lambda () '(0)))
    (ONE      ,(lambda () '(1)))
    ;; State
    (READY    ,(lambda () '(0)))
    (WAIT     ,(lambda () '(1)))
    ))

(define CMD.MAP
  `((v0  ,(lambda () (nat->bv 0 1)))
    (v1  ,(lambda () (nat->bv 1 1)))
    (CMD ,(lambda () '(cmd.0)))
    (p0  ,(lambda () '(I)))
    ))

(define STATE.SYMMAP
  `((wait ,(lambda () '(STATE)))
    (ready ,(lambda () '(!STATE)))
    ))
```

Figure 9: The file `rep.ss`

```
;;NEXT-STATE

(define next-state
   (lambda (s p0)
      (case s
         [ready (if p0 wait ready)]
         [wait  (if p0 wait ready)])))


;; NEXTSTATE.BIN.1

(define K
   (lambda (s p0)
      (case s
         [ready (if p0 wait ready)]
         [wait  (if p0 wait ready)])))


;; NEXTSTATE.BIN.2

((define STATE
    (lambda (s p0)
       (case s
          [ready (if I 1 0)]
          [wait  (if I 1 0)]))))


;; NEXTSTATE.BIN

((define STATE
    (lambda (s)
       (case s
          [ready (if I 1 0)]
          [wait  (if I 1 0)]))))
```

Figure 10: Specialization of the `next-state` signal

```
;; ENCODE.BIN1

(define CMD
   (lambda* ((s p0) v0 v1)
      (case s
         [ready v0]
         [wait (if p0 v0 v1)])))


;; ENCODE.BIN

((define cmd.0
    (lambda* ((s))
       (case s
          [ready 0]
          [wait (if I 0 1)]))))
```

Figure 11: Specialization of the `O` signal

.

```
;; DATAPATH.BIN
((O (SELECT STATUS 0 1))
 (STATUS (XPS STATE I)))

;; OUTPUT.BOOL
((O ((cmd.0))))

;; NEXTSTATE.BOOL
((STATE ((I & STATE) (I & !STATE))))

;; ENCODE.BOOL
((cmd.0 ((!I & STATE))))

;; NEXTSTATE.MIN
((STATE ((I))))

;; ENCODE.MIN
((cmd.0 ((!I & STATE))))

;; REGISTER
((STATE ((I))))

;; CONTROL
((cmd.0 ((!I & STATE))))

;; OUTPUT
((O ((cmd.0))))
```

Figure 12: Injection of derived implementation into logic synthesis.