Studies of the Single Pulser in Various Reasoning Systems

Steven D. Johnson,[1] Paul S. Miner,[2] and Albert Camilleri[3]

[1] Indiana University, Bloomington, IN 47405-4101 USA
[2] NASA Langley Research Center, Hampton, VA 23681-0001 USA
[3] Hewlett-Packard Company, Roseville, CA 95747-5596 USA

Abstract. The *single pulser* is a clocked sequential device which generates a unit-time pulse on its output for every pulse on its input. This paper explores how a single-pulser implementation is verified by various formal reasoning tools, including the PVS theorem prover for higher-order logic, the SMV model checker for computation tree logic, the DDD design derivation system, and the Oct Tools design environment. By fixing a single, simple example, the study attempts to contrast how the underlying formalisms influence one's perspective on design and verification.

KEYWORDS AND PHRASES: Formal methods, hardware verification, formal verification, theorem prover, higher order logic, model checker, design derivation, logic synthesis, PVS, SMV, CTL, DDD, Oct Tools.

1 Introduction

One of the problems confronting the transfer to practice of formal verification methodology is the confounding variety of formal systems available. This paper is an initial attempt to contrast several mechanized formalisms as applied to a simple common problem, a circuit called a *single pulser*. The single pulser is arguably the simplest sequential circuit that does anything interesting. Even so, its verification exposes interesting issues in each of the studies we have undertaken.

We look at four systems in this paper. The first and most general is the PVS theorem prover [11], which operates on higher-order logic expressions over inductive types. Next, we look at SMV, a symbolic model checker [10]. The propositional temporal logic on which SMV is founded is far less expressive than the higher-order logic of PVS; and consequently, the proofs are far more automatic. Third, we try DDD, a design derivation system based the algebra of first-order functional expressions [9]. Finally, we apply the Oct Tools suite of design synthesis tools [13] to the single-pulser problem.

As discussed in the following section, there are many ways of looking at a single pulser. The four studies we have done are not directly comparable because they address different facets of the verification task. However, there are common impressions resulting from these studies. In each case, there are conceptual, methodological, and representational questions to understand before the system can be applied. In our view, there is also evidence to support the need for more integrated reasoning environments, in which tools such as these can be used in consort to solve design problems.

Our goal in reporting these studies is to gain perspective on the variety of approaches that have been taken to formalize reasoning about hardware. We hope eventually to expand this project to include a greater number of systems. This work illustrates how design aspects are represented in different formal systems. It also conveys some of the intuitive feel of working within these systems. We feel that tutorial material such as this will help researchers and practitioners understand how reasoning tools relate to one another. More systematic and rigorous comparisons would certainly be valuable, but are well beyond the scope of these studies. Even if one were to have criteria for comparisons, a more diverse set of examples would be needed to make them meaningful.

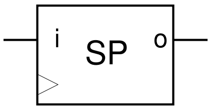
More detailed accounts of some of these studies (e.g. transcripts of interactions, script files, etc.) will be made available through [1]. Sections 3 through 6 contain references to more detailed descriptions of the individual systems under study. For a broader view of hardware verification systems and methods, Gupta's survey [7] contains an extensive bibliography and a partial taxonomy; it is a good starting point for further reading although it specifically mentions only one of the systems used in this work, SMV.

2 Informal Description of the Single Pulser

The *single pulser* comes from a textbook by Winkel and Prosser on clocked-synchronous design [12]. Their original English specification reads:

> **Problem Statement.** We have a debounced pushbutton, on (true) in the down position, off (false) in the up position. Devise a circuit to sense the depression of the button and assert an output signal for one clock pulse. The system should not allow additional assertions of the output until after the operator has released the button.

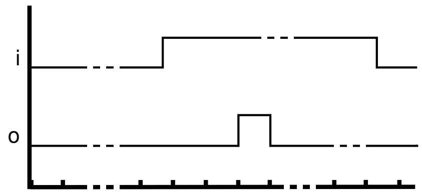
In all of the studies that follow, there is no attempt to account for debouncing an analog input signal; in fact, we shall also assume that input is synchronous. The single pulser device, *SP* has a one-bit input and a one-bit output:



Its observable behavior might stated as follows:

> *"SP emits a single unit-time pulse on o for each pulse received on i."*

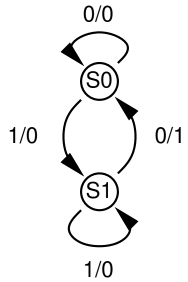
Actually, this sentence is an understatement of the specification, if formulated literally, although the necessary details would no doubt be inferred by a designer. The Winkel-Prosser problem statement indirectly says that there is exactly one output pulse for every input pulse. Furthermore, simplicity demands that the output pulse occur in the neighborhood of the input pulse (rather than occuring in independent periodic bursts, for example). Let us take the timing diagram below as a somewhat more rigorous informal specification.

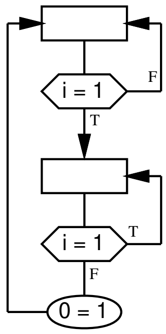


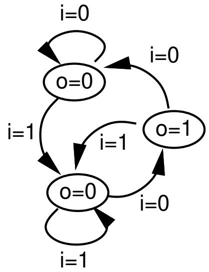
Ellipses indicate time intervals of undetermined duration. The left region of the diagram is intended to say that there are no extra output pulses and the right region that there is just one output pulse some time during every input pulse.

Any reasonable hardware implementation would pick either the beginning or the end of the interval to generate the output pulse. However, the diagram also admits some impossible implementations, such as one in which the output pulse occurs at the midpoint of the input pulse.

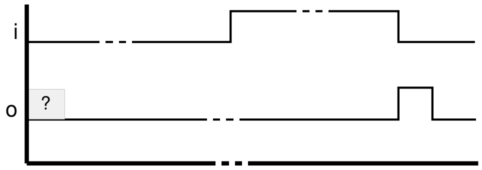
The *SP* timing diagram is like a *requirements specification,* describing the expected observable properties of the device. Similarly, could call one of the finite-state diagrams below a *design specification.* Each of them describes a synchronous process with, it is claimed, the required behavior.



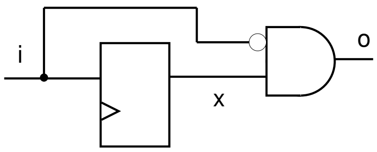




We show several kinds of state-machine diagrams because we are going to adopt perhaps the least familiar one, in the middle. It is the Algorithmic State Machine (ASM) diagram used by Winkel and Prosser. In ASM-diagrams, rectangles represent states, diamonds represent control-flow decisions, and ovals represent conditional (or "Mealy") outputs. There is also a convention that conditions occurring in the diagram are denied unless they are explicitly asserted; thus, "$o = 1$" is false (i.e., $o = 0$) on all but one of the paths. ASMs are relatively expressive in practice but more difficult to formalize. A timing diagram for the state machine is:

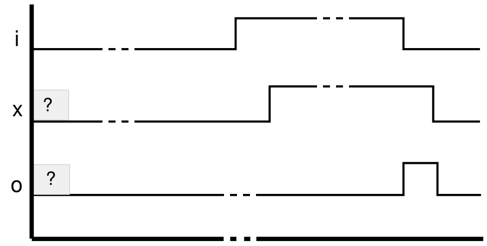


This diagram satisfies (or implements) the *SP* timing diagram, assuming that we don't care what happens at "time zero," and further, that the output pulse is sufficiently "within the neighborhood" of the input pulse.[4] An *implementation description* of the single pulser is represented by the circuit diagram:



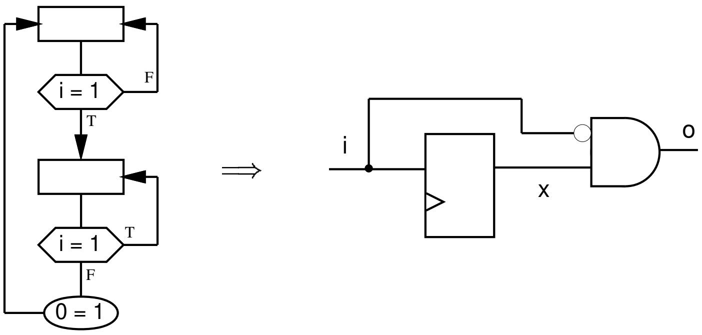
[4] The Reader is invited to jot down an appropriate definition of "neighborhood" at this point, as this is a topic we shall return to later.

It is easy to work out that this circuit has the timing diagram:

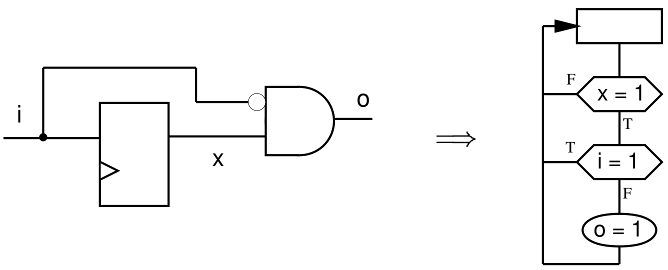


Hence, it is an implementation of both the design and requirements specifications.

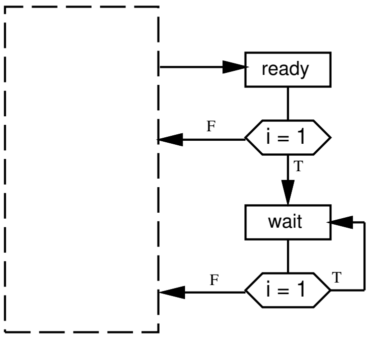
Winkel and Prosser make two additional observations about the single pulser which are relevant to its role as a tutorial example. The first of these remarks has to do with the duality of control and architecture in hardware descriptions. The *SP* circuit shown earlier is systematically derivable from the ASM specification:



However, now that we have the circuit we can look at it (specifically the delay element) as a piece of architecture under the control of the one-state (i.e., vacuous) ASM:



Similarly, the isolated view we have taken of the single pulser as a separate device overshadows another, equally valid, abstraction of *SP* as a synchronization protocol. "Being a single pulser" might be taken to mean that a larger circuit contains the same handshake as the abstracted single pulser:



We could also think of *SP* as being abstract with respect to the kinds of event sequences it recognizes as a "pulse." None of these generalizations arise in the studies we have carried out so far, but it might be good to keep them in mind.

3 Verification of a Single Pulser using PVS

PVS (Prototype Verification System) is a mechanical theorem proving system developed at SRI International [11]. Its specification language is based on simply-typed higher-order logic; it provides an interactive proof checker employing sequent calculus proof rules and it has decision procedures for linear arithmetic. One interacts with PVS through an ASCII text editor, but the system also provides formatting facilities for printed reports. Those facilities were used to generate the logic formulas in this section. The proof presented in this section is also discussed in [8].

Since the single pulser is a simple circuit, we bypass the state machine level and directly prove a candidate implementation with respect to a high-level specification. A plausible specification of a single pulser is the following (i and o‘ are functions from time to values, where time ranges over the integers and values are taken from $\{0, 1\}$. Variables m, n, j, and k are of type time):

$$
\begin{array}{l}
\mathrm{spec1}(i, o) : \mathrm{bool} \\
\quad = \\
\quad (\forall\, n, m : \\
\quad\quad \mathrm{Pulse}(i, n, m) \\
\quad\quad\quad \supset \\
\quad\quad\quad (\exists\, k : \\
\quad\quad\quad\quad n \le k \\
\quad\quad\quad\quad\quad \wedge\ k \le m \\
\quad\quad\quad\quad\quad\quad \wedge\ o(k) = 1 \\
\quad\quad\quad\quad\quad\quad\quad \wedge\ (\forall\, j : (n \le j \wedge j \le m \wedge o(j) = 1 \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \supset j = k))))
\end{array}
$$

An English reading of this formula is:

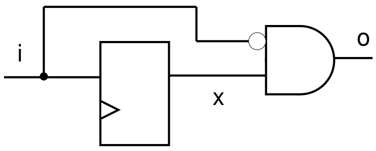
> Whenever there is a pulse on the input signal i, say from time n to time m, there is a unique time k in the vicinity of the input pulse when the output signal is asserted.

A graphical representation of $\text{Pulse}(f, n, m)$ is given by:

The definition of predicate Pulse in PVS is:

$$
\begin{aligned}
&\text{Pulse}(f, n, m) : \text{bool} \\
&\quad = (n < m \wedge f(n - 1) = 0 \wedge f(m) = 0 \\
&\qquad \wedge (\forall\, t : (n \leq t \wedge t < m \supset f(t) = 1)))
\end{aligned}
$$

Our candidate single pulser circuit is

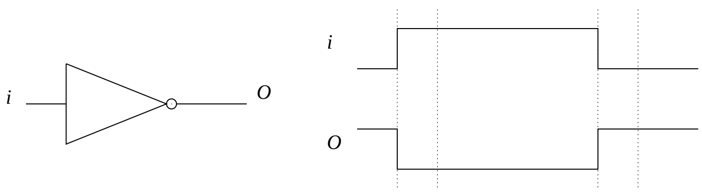


Using the style advocated by Gordon [6] we can represent this circuit as follows:

$$
\begin{aligned}
&\text{imp}(i, o) : \text{bool} = (\exists\, x : (\text{delay}(i, x) \wedge \text{and}^{\circ}_{\bullet}(i, x, o))) \\
&\text{delay}(i, o) : \text{bool} = (\forall\, t : (o(t + 1) = i(t))) \\
&\text{and}^{\circ}_{\bullet}(a, b, y) : \text{bool} = (\forall\, t : (y(t) = (-a(t)) \times b(t)))
\end{aligned}
$$

It is fairly simple to verify that this implementation satisfies spec1. A summary of the PVS proof is given in Figure 1. Each of the subgoal sequents displayed in Figure 1 can be easily verified using appropriate instantiations of quantified variables in the assumptions.

Unfortunately, spec1 is not sufficient. It only specifies the behavior of the circuit in the neighborhood of an input pulse. There are no constraints on the behavior of imp between pulses. As an example, a simple inverter satisfies our proposed single pulser specification. This is illustrated by



where the inverter is defined by

Verbose proof for `single_pulse1`.

`single_pulse1`:

$[1]\ (\forall\ (o, i : \text{signal}) : \text{imp}(i, o) \supset \text{spec1}(i, o))$

Expanding the definitions of spec1, imp, delay, and$\mathring{\bullet}$, −, Pulse
Applying (SKOSIMP*),
Instantiating the top quantifier in + with the terms: m!1
Applying propositional simplification and decision procedures,
which yields 2 subgoals:

`single_pulse1.1`:

$\{-1\}\ (\forall\ (t : \text{time}) : (x'(t+1) = i'(t)))$
$\{-2\}\ (\forall\ (t : \text{time}) : (o'(t) = (1 - i'(t)) \times x'(t)))$
$\{-3\}\ n' < m'$
$\{-4\}\ i'(n' - 1) = 0$
$\{-5\}\ i'(m') = 0$
$\{-6\}\ (\forall\ (t : \text{time}) : (n' \leq t \wedge t < m' \supset i'(t) = 1))$
$\{1\}\ \ o'(m') = 1$

`single_pulse1.2`:

$\{-1\}\ n' \leq j'$
$\{-2\}\ j' \leq m'$
$\{-3\}\ o'(j') = 1$
$[-4]\ \ (\forall\ (t : \text{time}) : (x'(t+1) = i'(t)))$
$[-5]\ \ (\forall\ (t : \text{time}) : (o'(t) = (1 - i'(t)) \times x'(t)))$
$[-6]\ \ n' < m'$
$[-7]\ \ i'(n' - 1) = 0$
$[-8]\ \ i'(m') = 0$
$[-9]\ \ (\forall\ (t : \text{time}) : (n' \leq t \wedge t < m' \supset i'(t) = 1))$
$\{1\}\ \ j' = m'$

Fig. 1. PVS proof trace for `single_pulse1`

$$\text{inv}(i, o) : \text{bool} \ = (\forall\ t : (o(t) = 1 - i(\ t)))$$

There are a number of ways to remedy this situation. One possibility is to modify our definition of Pulse to extend our notion of neighborhood until the beginning of the next pulse.[5] However, spec1 still does not constrain the behavior for all possible input streams. In the cases where the input has been high forever or remains high forever, the implementation is unconstrained. In addition to spec1, we need to show that the implementation also satisfies the following:

$$\begin{aligned}&\text{spec2}(i, o) : \text{bool}\\ &\quad =\\ &\quad (\forall\ k :\\ &\qquad o(k) = 1\end{aligned}$$

[5] See Footnote 4, in Section 2.

$$\supset \ \text{SinglePulse}(o, k) \\ \quad \wedge \ (\exists \, n, m : \ n \ \leq \ k \ \wedge \ k \ \leq \ m \ \wedge \ \text{Pulse}(i, n, m)))$$

This formula states that whenever the output is asserted, it is a single pulse and is in the vicinity of an input pulse. It is fairly simple to show that the output is a single pulse, if it is ever asserted. We can also easily show that the output is asserted after the input transitions from high to low. However, we have no means to show that the input pulse has been low previously. In other words, this circuit actually implements a synchronous falling-edge detector. It can only be a single pulser under the assumption that at some time in the past, the input was low. This is obviously a reasonable assumption; thus, we modify our verification condition by adding a clause about the environment in which the circuit is intended to operate, giving the following additional verification condition:

$$\text{env1}(f) : \ \text{bool} \ = \ (\forall \, t : \ (\exists \, t_1 : \ t_1 \ < \ t \ \wedge \ f(t_1) \ = \ 0)) \\ \text{single\_pulse2} : \ \text{LEMMA} \ \text{env1}(i) \ \supset \ (\text{imp}(i, o) \ \supset \ \text{spec2}(i, o))$$

The additional assumption about the environment requires some additional reasoning unrelated to the correct operation of the circuit. In particular, it was necessary to establish from env1 that a greatest such time exists.

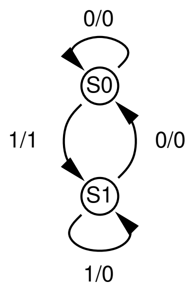
This completes our verification of the single pulser using PVS. Our example lacks some characteristics of larger verification efforts. In our proofs we expand the definitions of the circuit elements to their representation and invoke the built-in decision procedures of PVS. In a larger verification, we would prove properties about our representation and simplify accordingly, rather than employing the brute force approach shown here.

4 Verification of the Single Pulser Using Temporal Logic

We used the SMV (*Symbolic Model Verifier*) system [10] to specify and verify the single pulser circuit. An implementation is represented by a finite-state machine and the specification is represented by a formula in *computation-tree logic* (CTL). The system automatically verifies whether the given state-machine satisfies that formula, that is, provides a model which makes the formula true.

A CTL formula specifies a possibly infinite computation tree describing the intended behavior of a correct design [4, 3]. Universal and existential quantifiers, A and E, refer to paths in the tree. The modalities F (some future state), G (all states), X (next state) and U ("(strong) until," or an interval between states) refer to the totally ordered set of states along a path. The simple variables of a CTL formula are propositional with '`!`' ('`&`', '`|`', '`->`') standing for logical negation (conjunction, disjunction, implication). A state is specified by the propositions that hold in it.

An SMV description of the single pulser design is given by

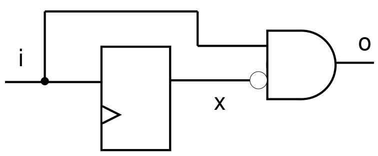



```
VAR state: {ready, wait};
    i : boolean;

ASSIGN
 init(state) := ready;
 next(state) :=
   case
    state = ready & !i : ready;
    state = ready &  i : wait;
    state = wait  &  i : wait;
    1                  : ready;
   esac;

DEFINE
  o :=
   case
    state = ready & !i : 0;
    state = ready &  i : 1;
    state = wait  &  i : 0;
    1                  : 0;
   esac;
```

Arbitrarily, this version of the state machine issues its output pulse at the rising edge of an input pulse rather than the falling edge as before. The state-machine description could be simplified to

```
ASSIGN next(state) := case i: wait, 1: ready; esac;
DEFINE o := (state = ready) & i;
```

(and even further once a boolean assignment is made for `ready` and `wait`) but we made a direct translation from the diagram, thinking that a more automatic system should tolerate a mechanistic approach. A representation of the implementation circuit is equally straightforward if we associate a state transition with a unit-time interval, so that a clocked register can be represented by the SMV `next` operator.




```
VAR
  x: boolean;

ASSIGN
  init(i) := 0;
  next(x) := i;

DEFINE
  rising_edge := i & !x;
```

The need to preset i corresponds to the initial-state assignment in the state machine. However, as was also the case with PVS, this reset assumption obscures some possibly significant anomalies that often occur during power-up intervals in physical circuits. We have immediately that the circuit implements the state machine. The keyword SPEC indicates an assertion to be verified.

```
SPEC  o = rising_edge
```

SMV confirms this to be true as it does with all of the SPEC statements that follow.

In order to express the more general single-pulser specification in SMV we must as before settle on an acceptable notion of *neighborhood* within which to look for an output pulse. In this case, let us define *neighborhood* to be the interval between two successive rising edges on i (Notice above that rising_edge refers not to the transition event on i but to the first unit-length period thereafter).[6] The single pulser specification decomposes into three overlapping properties as follows.

(a) `SPEC  AG (rising_edge -> (AF (o)))`
Whenever there is a rising edge o is 1 some time later.

(b) `SPEC  AG (o -> AX (A[!o U rising_edge]))`
Whenever o is 1 it becomes 0 in the next state (ie, one unit later) and it remains 0 at least until the next rising edge on i

(c) `SPEC  AG (rising_edge -> (!o -> (AX A[!rising_edge U o])))`
Whenever there is a rising edge, and assuming that the output pulse doesn't happen immediately, there are no more rising edges until that pulse happens. In other words, there can't be two rising edges on i without a pulse on o between them.

Property (b) is actually not valid because the semantics of the *until* operator would require rising_edge to become true on all paths—the so-called *strong until.* However, SMV has a provision called FAIRNESS which asserts that a condition holds infinitely often on all paths. If we assert FAIRNESS rising_edge then Property (b) becomes valid for the single pulser state machine.

We should comment on the difficulties and uncertainties we experienced in reaching this form of the specification. For the novice users, it took a long period to adjust to the expressive limitations of CTL, and it is still very much a matter of debate when a particular logical representation is "well put." Beginners quite often fell into logical traps, for example, writing formulas like AG(!i) -> AG(!o) to express, "In the case that i is never true, neither is o." This formula is true, but only because AG(!i) is a false premise. Coming up with the definition of

[6] This choice of the interval from rising edge to rising edge was based on the experience of the PVS study. See Footnote 5.

`rising_edge` seemed to be a key. Without this definition the CTL would be much more complex. Similarly, one should not try to say too much with a single sentence. Our specifications improved significantly once we decomposed the specification into a set of simpler properties.

Once the logical representations are determined, the proofs are immediate and automatic. Failed proofs produce scenarios that are useful in refining the formulations. However, in this study, invalid theorems always revealed inadequacies in the specification, rather than mistakes in the design.

5 Derivation of a Single Pulser using DDD

DDD (*Digital Design Derivation*) is a specialized transformation system for digital system design [9, 2]. It operates on two dialects of first-order functional expressions concretely represented by Scheme (Lisp) s-expressions. These dialects correspond to behavioral and structural forms of hardware description.

The goal of transformation is to reduce a higher-level algorithmic specification into a hierarchical network of processes. A *derivation* is a sequence of transformations and constructions applied to an initial expression, which is typically a design specification. Most DDD transformations preserve functional equivalence, although some are contingent on assumptions about the context (input-output timing, for example); and others add detail to the design (data representations, for example). As a formal object, a derivation together with any side conditions it synthesizes constitutes a proof of "implementation correctness," that is, equivalence between the design specification and its circuit implementation.

Typically, there are three phases in a derivation. The first is to manipulate the behavioral form to achieve certain architectural goals. The second is to construct and refine the structural expression describing that architecture. The third is to project the design to a concrete data representation, ultimately, a boolean system to which logic synthesis tools are applied.

In the single pulser study, we began with the specification `SP` shown in Fig. 2. `SP` is a proper Scheme program in which the states of the single pulser ASM represented by a system of tail-recursive functions definitions, `READY` and `WAIT`. This is a standard way to model finite-state control with function expressions. Within each function definition, the *let*-bindings for `X` and `I` provide a simple input/output interface: `SP` executes as a Scheme program to animate the specification. Thus, the DDD formulation differs from predicate formulations in that the expressions on which it operates can be directly executed to explore the design behavior. However, the I/O supporting animation contaminates the expressions when considered as a hardware description. The algebra used to transform these expressions into hardware can also be used to isolate and factor out the modeling interface, as is demonstrated in this study.

```
(define SP
(lambda (IN)
  (letrec
    ((READY (lambda (O)
       (let ((X (out O))
             (I (= (inp IN) 1)))
            (if I
                (WAIT 0)
                (READY 0)))))

     (WAIT  (lambda (O)
       (let ((X (out O))
             (I (= (inp IN) 1)))
            (if I
                (WAIT 0)
                (READY 1))))))

      (READY 0))))
```

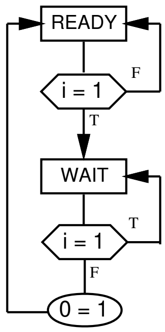
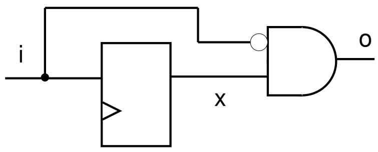


Fig. 2. DDD design specification of a single pulser

A second dialect of functional modeling expression is used to describe circuit structure. The target implementation would be:

```
(define SP
(lambda (I)
  (system-letrec

    ((X  (! 0 I))
     (O  (ANDx I X)))

     O)))
```



The `system-letrec` expression defines a network of non-terminating *streams* representing infinite sequences of values over time. The defining expression for `X` uses DDD's delay operator, '`!`" to (arbitrarily) initialize sequence `X` with a zero. Sequence `O` is obtained by extending the binary operation `ANDx` element-wise to the streams `I` and `X`. Recursion in `system-letrec` expressions corresponds to feedback in the stream network, but there is no feedback in the single pulser example. Streams are not standard Scheme constructs but are added as a syntactic extension [2].

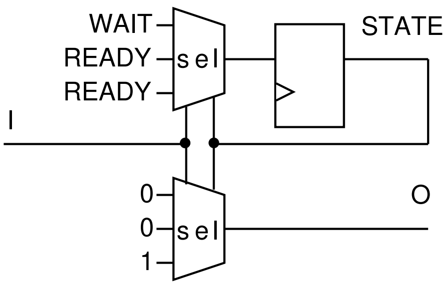
The first step in the derivation constructs an initial structural description. The two components of this description are (a) a *selection combination*, which represents the control structure of the specification, and (b) an initial sequential system.

```
(a) (define select
      (lambda* ((s p0) v0 v1 v2)
        (case s
          [READY (if p0 v0 v1)]
          [WAIT (if p0 v0 v2)]))))
```

```
(b) (define SP.1
     (lambda (IN)
      (system-letrec
        ((STATUS (XPS STATE I))
         (I      (SELECT
                   STATUS
                   (= (INP IN) 1)
                   (= (INP IN) 1)
                   (= (INP IN) 1)))
         (STATE  (! READY (SELECT STATUS WAIT READY READY)))
         (O      (! 0 (SELECT STATUS 0 0 1)))
         (X      (SELECT STATUS (OUT O) (OUT O) (OUT O))))
        (list STATUS I STATE O X))))
```

Essentially, the streams in `SP.1` are execution traces of the formal parameters in specification `SP`. The `STATE` and `STATUS` streams, together, implement a control automaton. The `select` combinator contains one other extension of standard Scheme syntax: nested lambda parameters such as `((s p0) v0 v1 v2)` make it easier to manipulate multiple-output functions.

The next step is to isolate the single pulser circuit from the artifacts of the modeling interface, namely, the signal `X`, the subexpression `(= (INP IN) 1)`, and the "register" represented by the '`!`' in the defining expression for `O`. In order to isolate the `O` register, we had to expand '`O`' by its defining expression in the equation for `X`, and then identify the common subexpression `(SELECT STATUS 0 0 1)`. Once this was done, DDD was instructed to partition `SP` into two subsystems, one of which encapsulated the modeling interface. After a term simplification, the residual subsystem is the pure single pulser:



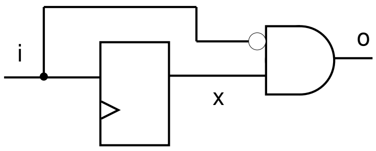

```
(c) (define SP.1.1
     (lambda (I)
```

```
(system-letrec
  ((STATUS (XPS STATE I))
   (STATE (! READY (SELECT STATUS WAIT READY READY)))
   (O (SELECT STATUS 0 0 1)))
  (XPS STATUS STATE O))))
```

Once binary values are assigned to WAIT and READY, this system can be projected to logic synthesis. DDD expands and simplifies the occurrence of SELECT in the defining expression for STATE to get:

```
(define select-state
   (lambda* (p0) (if p0 wait ready)))
```

Thus, if we choose WAIT $= 0$ and READY $= 1$ then STATE simplifies to I. Under this state assignment, the Oct Tools logic minimizer also finds a single-gate realization of (SELECT STATUS 0 0 1), and we end up with the desired circuit:



In all, six DDD commands were involved in the derivation, plus a rather cumbersome script of logic-synthesis commands.

6 Synthesis of a single-pulser using Oct Tools

A standard CAD system—we use Oct Tools [13] as a readily available example—can automatically synthesize efficient implementations of the single-pulser. We shall not pursue whether a set of CAD tools should be classified as a formal system, but it is software that supports a specific kind of reasoning. This point is discussed further at the end of this section.

Within the Oct Tools environment are two hardware description languages, a behavioral HDL called *Bdsyn* and a structural HDL called *Bdnet*. Figure 4 shows the control-specification fragment of the single-pulser in *Bdsyn*. An affiliated *Bdnet* structural description, not shown, specifies the kind of storage device used to hold pres_state, among other things.

From this description, an intermediate boolean system description (blif file) was generated (using the bdsyn/bdnet translation tools). A minimizer and technology mapper were applied to the intermediate file (using misII). Physical placement (octflatten) and routing (wolfe) tools were then used to create the VLSI layout shown in Fig. 3. When targeted to standard-cell technology, this process resulted in an implementation containing just one flipflop and one gate, as we would hope. There was no manual intervention beyond writing the specification and choosing the appropriate synthesis libraries and tactics. A *Bdsyn/Bdnet* description of the stateless version of the single-pulser (discussed in Sec. 2) resulted in an apparently identical VLSI layout, as did Oct Tools synthesis of the implementation derived in DDD.

The *Bdsyn* specification in Figure 4 includes an explicit assignment of boolean values to the state labels `READY` and `WAIT`. This assignment did affect the standard-cell layout. The opposite assignment `READY` $= 1$, `WAIT` $= 0$ introduced an inverter, which did not change the net number of transistors in the realization but did enlarge the layout area by 12%. The inverter might have been eliminated by a re-timing tool, but this avenue was not explored.

We believe it would be hard to differentiate between the kind of user-tool negotiation employed in the previous paragraph, and that seen when using (say) a general purpose theorem prover. For this reason, we are inclined to include a CAD environment among the other reasoning systems we are exploring, even if it does broaden the sense of the term "reasoning."

7 Conclusion

The four studies presented in this paper address distinct aspects of the verification task, as illustrated in Fig. 5. They involved four modes of description, three of which are formal. We likened the single-pulser timing diagram to a true specification because it details the observable properties of interest. The state-machine representation should be called a design description; among other things, it determines the particular wave form used to satisfy the specification. The circuit diagram, then, is an implementation description, detailing how this wave from is generated. Finally, there is a realization of the implementation in the form of a VLSI layout.

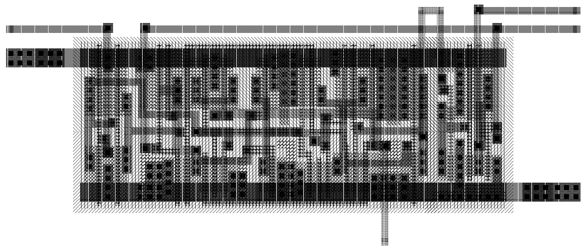


Fig. 3. Oct Tools synthesized layout of the single pulser as displayed by vem

```
MODEL SP

! OUTPUTS next_state<0>, out<0>
=
! INPUTS pres_state<0>, in<0>;

CONSTANT
        READY = 0, WAIT = 1,
        FALSE = 0, TRUE = 1;

ROUTINE single_pulser_control;
  next_state = READY;
  out = FALSE;
  SELECT pres_state FROM
    [READY]: BEGIN
      IF in EQL TRUE THEN
        BEGIN
          next_state = WAIT;
        END ELSE BEGIN
          next_state = READY;
        END;
      END;
     [WAIT]: BEGIN
      IF in EQL TRUE THEN
        BEGIN
          next_state = WAIT;
        END ELSE BEGIN
          next_state = READY;
          out = TRUE;
        END;
      END;
  ENDSELECT;
ENDROUTINE;
ENDMODEL;
```

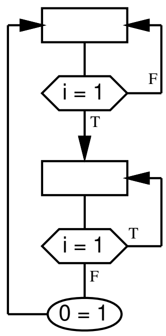


Fig. 4. SP in *Bdsyn*

Although diagrams are used informally in this paper, efforts are underway to make their meaning precise, with the goal of extending formal-reasoning systems to be more visually oriented [8, 5].

In PVS, the most general of the formal systems used, a proof directly establishes that the implementation satisfies the requirements of the specification. The intermediate representation of the state machine was not needed, although it, too, could have been formalized in PVS. A more complicated proof exercise

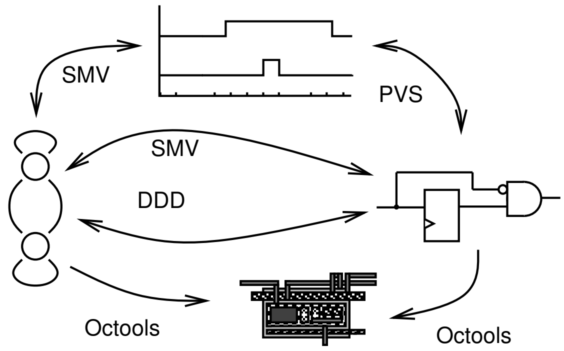


Fig. 5. Relating the studies

in PVS would typically use the design description. We had difficulties getting the specification right. It was never difficult to prove that a correct circuit met the specification, but the proof process exposed pathologies that would make unacceptable circuits correct. In particular, our notion of *neighborhood* had to be strengthened before we could reject unacceptable implementations, and we had to explicitly exclude the pathological case of an infinite pulse.

In the SMV study, temporal logic formulas were found to represent the timing-diagram specification, and SMV definitions and assignments were used to represent the state machine and circuit descriptions. We made many unsuccessful attempts at specifying a single pulser's behavior. As was the case in the PVS study, a bad specification usually resulted in a valid but inadequate theorem. In other words, SMV would affirm that the implementation was correct, but cross examination showed that the specification was too weak or simply wrong. Typically, our failures abused implication in some way, and the model checker would find a path that falsified the premise. The limited expressiveness of CTL made it more challenging to produce an acceptable form of the specification, but the proofs were always automatic and immediate.

DDD applies to the problem of deriving an implementation from a design. In the DDD environment, the boolean optimization is done by logic synthesis tools. Since the DDD algebra is specialized for manipulating sequential system descriptions, we did not encounter the kinds of logical pathologies found in PVS and SVM. The derivation study actually focused on isolating the design description from its interface to a modeling environment. This was a good exercise of the algebra, and also showed how other modeling tasks can interfere with formal reasoning processes. Nevertheless, We had to use some very round-about tactics to get the DDD algebra to do exactly what we wanted.

Synthesis of the single pulser was straightforward for an experienced Oct Tools user. Experimenting with minor variations, such as changing the state assignment, led to a pattern of reasoning that we found quite similar to the interaction with the other systems.

In each of the studies, application of the tool is complicated by some aspect of problem representation, but it is a different aspect in each case. In PVS a refinement of environmental constraints—no infinite pulses—was needed to carry the proofs through. In the SMV study, there were similar problems with pathalogical cases, and while the proofs were automatic, we found it hard to represent the specification in the less expressive CTL. In DDD, the modeling context interferred, as discussed earlier. Finally, the complexity in using a design synthesis environment lies in understanding what tools to apply and when to be satisfied with the outcome.

We plan to continue collecting and comparing studies of the single pulser, and will maintain the results in [1].

8 Acknowledgments

We are especially grateful to Shyamsundar Pullela for his contributions to earlier versions of this paper, and to both Shyam and Kathi Fisler for their help in the SMV study. Kamlesh Rath has participated in many of the discussions leading to this paper and provided expertise in the use of Oct Tools. Esen Tuna participated in the development of the DDD derivation of the single-pulser. The original PVS study was done during discussions of visual inference with Jon Barwise, Gerry Allwein, and Kathi Fisler. Zheng Zhu has made several valuable comments about these studies and earlier drafts of this paper.

This research was supported, in part, by the National Science Foundation under grant number MIP92-08745. We are also grateful to the National Aeronautics and Space Administration for granting leave to Paul Miner to pursue graduate study at Indiana University.

References

1. Further details about these studies can be obtained through the World Wide Web via URL `www.cs.indiana.edu`. Access the *Single Pulser Study* through the *Hardware Methods Group* thread in the list of departmental research projects. Individuals wishing to contribute to this collection should contact `sjohnson@cs.indiana.edu` or write Hardware Methods Laboratory, Indiana University Computer Science Department, Bloomington Indiana, USA.
2. Bhaskar Bose. DDD - A Transformation system for Digital Design Derivation. Technical Report 331, Computer Science Dept. Indiana University, May 1991.
3. J.R. Burch, E.M. Clarke, D.L. Dill, and K. L. McMillan. Sequential circuit verification using symbolic model checking. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, June 1990.

4. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specification. *ACM Transactions on Programming Languages and Systems*, 8(2), April 1986.
5. Kathi Fisler. Extending formal reasoning with support for hardware diagrams, 1994. This volume.
6. M.J.C. Gordon. Why higher order logic is a good formalism for specifying and verifying hardware. In G.J. Milne and P.A. Subrahmanyam, editors, *Formal Aspects of VLSI Design: Proceedings of the 1985 Edinburgh Conference on VLSI*, pages 153–177. North Holland, 1986.
7. Aarti Gupta. Formal hardware verification methods: A survey. *Formal Methods in System Design*, 1:151–238, 1992.
8. Steven D. Johnson, Gerard Allwein, and Jon Barwise. Toward the rigorous use of diagrams in reasoning about hardware. In Gerard Allwein and Jon Barwise, editors, *Working Papers on Diagrams and Logic.* Indiana University Logic Group *Preprint* IULG-93-24, May 1993.
9. Steven D. Johnson and Bhaskar Bose. A system for mechanized digital design derivation. In P. A. Subramanyahm (ed.), Participants' procedings of the *ACM/SIGDA International Workshop on Formal Methods in VLSI Design,* Miami, Florida, USA, January 1991., December 1990. With appendix as Indiana University Computer Science Department Technical Report No. 323.
10. K. L. McMillan. *Symbolic Model-Checking: An Approach to the State Explosion Problem.* PhD thesis, Carnegie Mellon University, 1992.
11. S. Owre, J.M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer Verlag.
12. Franklin Prosser and David Winkel. *The Art of Digital Design.* Prentice-Hall, Englewood Cliffs NJ, second edition, 1987.
13. Rick L. Spickelmier. *Release Notes for Oct Tools Distribution 5.1.* Electronics Research Laboratory, University of California, Berkeley, August 1991.