Modeling Transistors Applicatively[1]

by

Steven D. Johnson and C. David Boyer
Computer Science Department
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 253

Has appeared in *IFIP WG10.2 International Workshop on The fusion of Hardware Design and Verification, Glasgow, Scotland, July 4—6, 1988,* (Edited by G.J. Milne) Elsevier Science Publishers, pages 397—420.

Modeling Transistors Applicatively[1]

by

Steven D. Johnson and C. David Boyer
Computer Science Department
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 253

Has appeared in *IFIP WG10.2 International Workshop on The fusion of Hardware Design and Verification, Glasgow, Scotland, July 4—6, 1988,* (Edited by G.J. Milne) Elsevier Science Publishers, pages 397—420.

Modeling Transistors Applicatively*

Steven D. Johnson and C. David Boyer

Computer Science Department
Indiana University
Bloomington, Indiana, U.S.A

An applicative modeling language is used to explore logical transistor behavior. An applicative source language is good at describing functional qualities, but relational qualities require a careful treatment. With this issue in mind, we examine ways to represent bidirectional information flow in combinational systems. Formal approaches to hardware verification and synthesis enjoy the support of flexible symbolic processing systems, which readily represent disparate aspects of design. What we mean here by "modeling" is the *ad hoc* manipulation of such representations. Our main purpose is to explore and illustrate techniques for this kind of activity. In particular, we show how the interrelated qualities of lazy semantics and recursive data definition contribute to the natural description of hardware structures and behaviors.

1. Introduction

Hardware logics project electronics into simpler formal systems in order to facilitate reasoning. Unlike simulators, logics do not fix a ground interpretation; they adapt to disparate levels of description. Hence, their mechanical support must provide highly flexible underlying representations. Mechanized logics are also programming languages for their symbolic processing subsystems. *Symbolic modeling* is the construction and manipulation of data structures representing some aspect of a design. It is analogous to building a scale model to explore certain properties of a mechanical design. This differs from the task of defining formal hardware models, say for the purpose of hardware verification. It is an engineering activity, *ad hoc* by nature, focusing on arbitrary facets of a design.

*Research reported herein was supported, in part, by the National Science Foundation under grants numbered MIP 87-07067 and DCR 85-21497

This paper examines techniques for representing low levels of digital hardware in an applicative modeling language. Modeling methodology has potentially profound implications for engineering practice and design automation. However, its techniques are influenced by the still evolving semantics of the languages used. In the case of functional languages, a "lazy" semantics promotes different representation choices than would an applicative-order dialect. In particular, the facility for recursive data definition seems fundamental to the natural description of hardware. Modeling sequential behavior with reflexive stream-like objects is widely discussed [9, 11, 16, 2, 18]. Representing other aspects of design are less commonly understood.

We explore frontiers of applicative modeling through various treatments of logical transistor behavior. It is a good test of how this class of notation fares at its descriptive boundaries. Applicative notation is inherently directional, making it difficult to address *bidirectional* qualities.

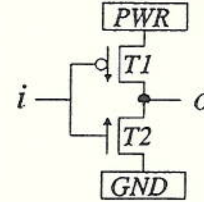
We have a theme: exploring symbolic hardware modeling in an applicative surface language; and a plot: representing bidirectional behavior; so perhaps a moral could be added: *Isolated qualities are best dealt with in their own terms.* Since functionality is a prominent abstraction in digital design, applicative notation will be prominent in digital description. Of course, this does not justify a *purely* functional view of hardware. The need for more generality is evident in electrical engineering and in verification. Hanna and Daeche note that the "functional subset of [typed higher-order logic]" provides an *animated* design notation [8] through evaluation. If this quality is to be fully exploited, a modeling methodology must develop for it.

Bidirectionality alone is not sufficient cause to abandon an applicative modeling framework. It can be dealt with at modest expense, as measured in syntax. This fact is illustrated in a series of treatments of a small CMOS circuit. Though there is a sense of refinement from one example to the next, each treatment reflects a reasonable engineering view. The level of modeling, based on a four valued domain of voltage values, extends to the more detailed levels now addressed in verification research.

1.1. Related Work

Consider the CMOS inverter depicted to the right below. Given terms Pwr and Gnd for power and ground, functions P and N for transistors, and a binary *join*-operation $\bullet$, an applicative inverter description is

$$INV(i) \stackrel{\text{def}}{=} P(i, \mathsf{Pwr}) \bullet N(i, \mathsf{Gnd})$$



In a relational notation, an analogous expression is

$$INV(i, o) \equiv N(i, \mathsf{Pwr}, o) \bullet P(i, \mathsf{Gnd}, o)$$

where the '•' is a meta-operator making the subterms coincide, in some semantic sense, on the variables i and o.

The inverter's directionality is implicit in its applicative description. In the relational notation it is not, and an explicit distinction might be needed to relate $INV(i, o)$ and $INV(o, i)$. An example is Gordon's predicate formulation [7], also discussed by Winskel [20] and Musser [15]. Other calculi on relational notation, such as Milne's [14] and Gopalakrishnan's [6], can specify the directionality.

Though most circuit designs are globally directional, this property is not always inherited by their components. A popular example is a CMOS full adder presented by Gordon [7, 15]. That circuit inspired the smaller one in Figure 1, which is used as an example in this paper.

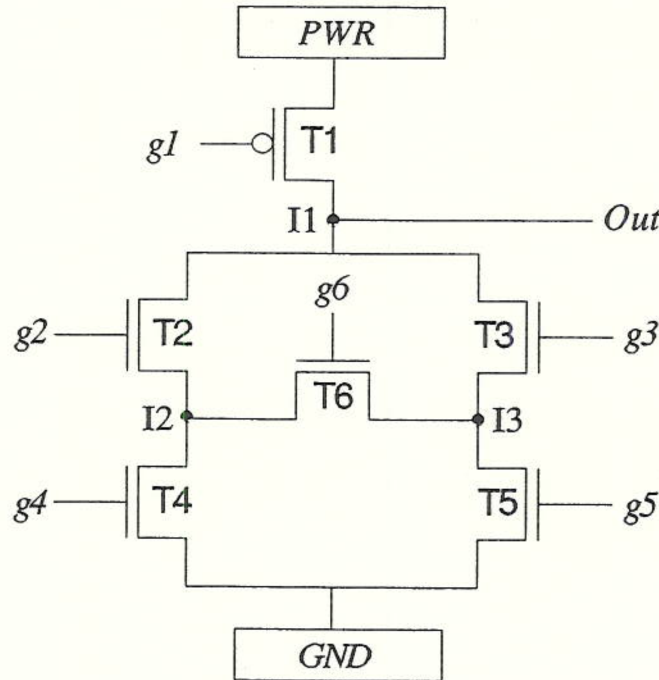


FIGURE 1 – CKT, a CMOS circuit with a bidirectional transistor

This network, called CKT, is easily expressed in relational terms and hard to express as an applicative combination. Its transistor T6 must be regarded as bidirectional. In addition, CKT does not work for all configurations of its gates.

Assuming we always want a high or low voltage at the point Out, and with high voltage interpreted as true we should have

$$g_1 \equiv (g_2 \wedge g_4) \vee (g_2 \wedge g_6 \wedge g_5) \vee (g_3 \wedge g_6 \wedge g_4) \vee (g_5 \wedge g_3).$$

Otherwise, Out is either shorted or not powered. These four conditions, powered high, powered low, shorted, and unconnected, are natural base values for reasoning about combinational MOS designs. They arise, or can be instantiated, in all of the recent formal models for VLSI verification (e.g. [20, 15, 4, 2]).

In evaluating CKT, an engineer projects the schematic to logic in a number of ways for verification purposes. The boolean function of the circuit is determined as the disjunction/conjunction of the gates along parallel/serial paths from output points to voltage supplies. Another analysis verifies the electrical integrity of the circuit, as suggested by the formula above. Once satisfied with the behavior at this simple level, more elaborate electrical reasoning may come into play, involving strength, capacitance [20, 15, 5] and temporal abstraction [13, 2].

2. Applicative Modeling

Functions are easier to reason about than relations because they enjoy more properties. To say that the two-place function f is associative, one writes

$$f(x, f(y, z)) = f(f(x, y), z).$$

The corresponding assertion about a relation R might be written,

$$R(y, z, U) \ \& \ R(x, U, V) \quad \textit{iff} \quad R(x, y, W) \ \& \ R(W, z, V)$$

f's functionality permits us to refer to its "output" uniquely as $f(x, y)$ and hence to employ equality in the assertion.

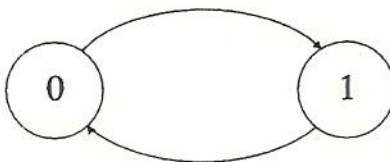
Unfortunately, one cannot freely use equality in applicative modeling. Functional programming is based on solving simultaneous systems of equations by inductive approximation. The space of values is structured to make this possible. This matter is explained fully in text books on functional programming (e.g. [10]). Systems with defined solutions include enough base information to build them inductively. One kind of example is a system of function definitions, such as

```
F(n) where
  F(x) = if zero?(x) then 0 else G(decrement(x))
  G(x) = if zero?(x) then 1 else F(decrement(x))
```

By induction, F and G both are defined for all integers $n \geq 0$. A second kind of example is a recursive system of data definitions, such as

```
A where
  A = [0 ! B]
  B = [1 ! A]
```

The expression [e_0 ! e_1] forms a *list,* whose head is e_0's value and whose tail is e_1's value. Again by induction, the nth elements of lists A and B are all defined. A may be thought of as an infinite list, [0, 1, 0, 1, $\cdots$], representing sequential behavior over time. In a graph processing system, A also denotes a cyclic graph,



It is natural to represent circuit networks in this way. O'Donnell gives several examples, using recursive data declarations to build and manipulate descriptions of hardware [16].

It is expected that the reader has some familiarity with applicative programming. The programs in this paper, including those above, have a syntax similar to that of applicative languages now in use [10, 17]. They were mechanically translated into a more primitive language, called *Daisy;* details are given in [3]. Daisy is related to Pure Lisp. It is a general purpose, purely applicative, lazy, list processing language [12]. There is no specialization for hardware description. Primitive operations, always lower case and prefix, are explained as they arise.

Daisy's symbolic basis includes numbers, literal symbols, and binary graph cells. The *list expression* [A, B, C] forms a list [α, β, γ], containing the respective values of subexpressions A, B, and C. The *head* of this list is α and its *tail* is the list [β, γ]. List concatenation is expressed with an exclamation point; instead of [A, B, C], one could write [A ! [B ! [C ! [ ]]]].

Four symbolic voltage values (Recall Section 1.1) are represented by the literal symbols H (connected to high voltage), L (connected to low voltage), Z (not connected), and X (connected to high and low, i.e., shorted). Identifiers PWR and GND are defined for the power and ground supplies.

```
def PWR = "H"
def GND = "L"
```

Tests H?, L?, and so on, are coded for each value.

```
    |
    | H?, L?, Z?, X? : Value --> TruthValue
    |
def H?(V) = same? [V, "H"]
def L?(V) = same? [V, "L"]
def Z?(V) = same? [V, "Z"]
def X?(V) = same? [V, "X"]
```

The '|' character starts a comment; the type assertions are not checked. Same? is a primitive test for equality. The function H? tests whether its argument is the literal H.

The '•' function combines two values according to the table on the right. It is implemented by the program DOT, written to emphasize symmetry.

```
    |
    | DOT: [Value, Value] --> Value
    |
def DOT [U, V] =
  if OR [Z?(U), X?(V), same?[U, V]]
     then V else
  if OR [Z?(V), X?(U), same?[V, U]]
     then U
  else "X"
```

•	Z	H	L	X
Z	Z	H	L	X
H	H	H	X	X
L	L	X	L	X
X	X	X	X	X

OR is the disjunction of a list of tests.

```
    |
    | OR: [ TruthValue ... TruthValue ] --> TruthValue
    |
def OR [] = FALSE
    OR [TV ! Etc] = if TV then TRUE else OR(Etc)
```

The type declaration [TruthValue ... TruthValue] means a list of zero or more truth values—again, these are just comments. A simple form of sequential, pattern oriented binding is used [17]. An equivalent definition of OR is

```
def OR(TVs) = if nil?(TVs) then FALSE else
              if head(TVs) then TRUE
              else OR(tail(TVs))
```

The function DOTs returns the DOT-product of a list of values.

```
       |
       | DOTs: [ Value ... Value] --> Value
       | ACCUMULATE:[ Value, [Value ... Value]] --> Value
       |
   def DOTs (Vs) = ACCUMULATE ["Z", Vs]
     where
       ACCUMULATE [ V, []] = V
       ACCUMULATE ["X",  Vs] = "X"
       ACCUMULATE [ U, [W ! Ws]] = ACCUMULATE [DOT[U, W], Ws]
```

ACCUMULATE's first argument is the accumulating value, which DOTs initializes to Z. Values from the list Vs are incorporated until the list is exhausted or an X is discovered.

Throughout this paper, assume that let-expressions and where-expressions are recursive, even though not all of them have to be.

In Daisy there is an *indeterminate* list constructor which specifies content but not order. Let α and β denote the values of expressions A and B, respectively. The *multiset* expression

$$\{A,\ B\} \text{ returns } \begin{cases} [\alpha,\ \beta] & \text{provided } \alpha \text{ exists, or} \\ [\beta,\ \alpha] & \text{provided } \beta \text{ exists.} \end{cases}$$

If both α and β exist, either list might be returned. Multisets are concurrency constructs. A and B are evaluated concurrently, and the resulting list is ordered in a demand driven fashion as the computations terminate. The expression head$\{E_1, \cdots, E_n\}$ chooses the cheapest of the n alternatives. Computation ceases once the head of the list is determined. In particular,

$$\text{head}\{A,\ B\} \text{ returns } \begin{cases} \alpha & \text{if } \alpha \text{ exists and } \beta \text{ does not, and} \\ \beta & \text{if } \beta \text{ exists and } \alpha \text{ does not.} \end{cases}$$

The OR function applied to a multiset, OR$\{E_1 \cdots E_n\}$, returns true should *any* one of E_is produce true, even if others diverge.

Multisets are used to implement a *guarded choice* construct used in Section 4. Below, the function GUARD looks at Test and returns Value or the reserved symbol QUIT should the test fail. GIF is a variation of OR that skips QUIT rather than FALSE. The *error* case will not occur in any of the examples.

```
       |
       | GUARD : [TruthValue, Value] --> (Value + "QUIT")
       |
   def GUARD [Test, Value] = if Test then Value else "QUIT"
```

```
    |
    | GIF: [(Value + QUIT) ... (Value + QUIT)] --> Value
    |
def GIF [] = "error"
    GIF ["QUIT" ! Etc] = GIF(Etc)
    GIF [Value ! Etc] = Value
```

Applied to a multiset of GUARD-expressions, the tests are made concurrently, so that

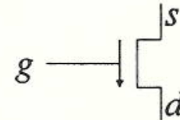
```
GIF { GUARD [Test, Value]
          ⋮
      GUARD [Test, Value]
    }
```

returns some value whose test holds.

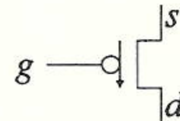
3. Directional Modeling

The most immediate applicative treatment of a transistor is as a function from a source and gate to a drain. Functions for P-type and N-type transistors are defined below [NOTE 1]. These differ only in their interpretation of the gate. Depending on the value at the gate g, the drain d of a transistor is either not connected or takes its value from the source s. A transistor is sometimes drawn with an arrow to show it's direction.

```
def NT [g, s] = d
  where
    d = (if H?(g) then s else "Z")
```

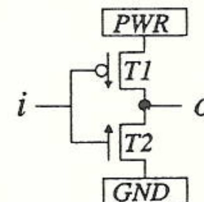



```
def PT [g, s] = d
  where
    d = (if L?(g) then s else "Z")
```



A circuit with a direction, such as the inverter discussed earlier, is an applicative combination of directional transistors [NOTE 2]:

```
def INV(i) = o
   where
       o = DOT [T1d, T2d]
     T1d = PT [i, PWR]
     T2d = NT [i, GND]
```



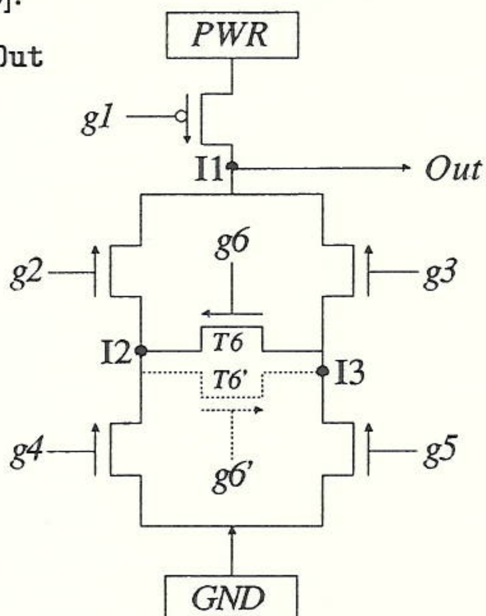
Extra variables are introduced in correspondence with the schematic. Identifiers T1d and T2d refer to the transistors' d-outputs. The equation for o joins these two outputs.

A simple approach to describing CKT (Figure 1) is to code one instance of transistor T6 for each of its directions [NOTE 3].

```
def CKT [g1, g2, g3, g4, g5, g6] = Out
  where
    Out = I1

     I1 = DOT [T1d, DOT [T2d, T3d]]
     I2 = DOT [T4d, T6d]
     I3 = DOT [T5d, T6d']

    T1d = PT [g1, PWR]
    T2d = NT [g2,  I2]
    T3d = NT [g3,  I3]
    T4d = NT [g4, GND]
    T5d = NT [g5, GND]
    T6d = NT [g6, T5d]
   T6d' = NT [g6, T4d]
```



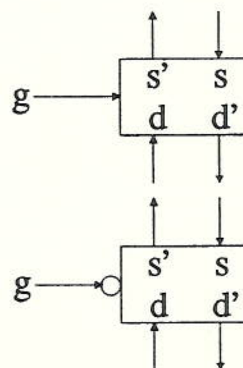
The equations for I1, I2, and I3 define internal junctions in the circuit as labeled in the schematic. This certainly works; the program computes this table of values for H and L combinations of gate inputs.

Out = I1	←			g_4, g_5, g_6				→
g_1, g_2, g_3	LLL	LLH	LHL	LHH	HLL	HLH	HHL	HHH
LLL	H	H	H	H	H	H	H	H
LLH	H	H	X	X	H	X	X	X
LHL	H	H	H	X	X	X	X	X
LHH	H	H	X	X	X	X	X	X
HLL	Z	Z	Z	Z	Z	Z	Z	Z
HLH	Z	Z	L	L	Z	L	L	L
HHL	Z	Z	Z	L	L	L	L	L
HHH	Z	Z	L	L	L	L	L	L

However, the two instances of T6 destroy a desired correspondence between the description and the actual circuit. CKT has six transistors, not seven. Such redundancy, if used, should at least be uniform. Therefore, let us encapsulate two directions of information flow in *all* transistors.

```
def NT [g, s, d] = [s', d']
  where
    s' = if H?(g) then d else "Z"
    d' = if H?(g) then s else "Z"

def PT [g, s, d] = [s', d']
  where
    s' = if L?(g) then d else "Z"
    d' = if L?(g) then s else "Z"
```

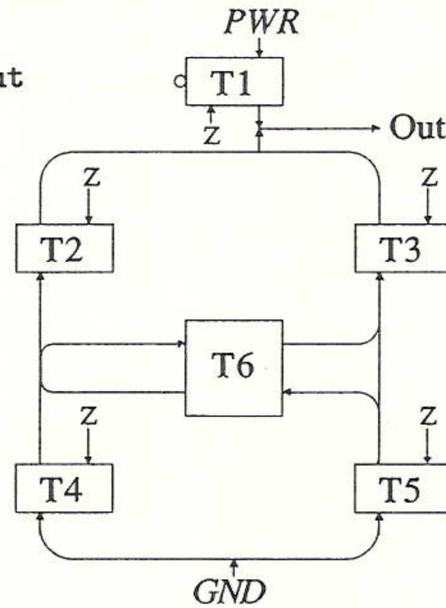


A convention, used here and later, places primes on out-going information. Gates are still treated as pure inputs. Returning an output g' does not contribute anything at this level of modeling, and the notational overhead is already bad enough. CKT is now described as follows.

```
def CKT [g1, g2, g3, g4, g5, g6] = Out
  where
    Out = I1

    I1 = DOTs [T1d, T2d, T3d]
    I2 = DOT [T4d, T6d]
    I3 = DOT [T5d, T6s]

    [T1s, T1d] = PT [g1, PWR, "Z"]
    [T2s, T2d] = NT [g2,  I2, "Z"]
    [T3s, T3d] = NT [g3,  I3, "Z"]
    [T4s, T4d] = NT [g4, GND, "Z"]
    [T5s, T5d] = NT [g5, GND, "Z"]
    [T6s, T6d] = NT [g6, T5d, T4d]
```



The proliferation of variables could be reduced by replacing the formal pairs (e.g. [T4s, T4d]) by a single identifier (e.g. T4) and coding selector functions (e.g. drain(T4)) on the right. However, this is not the only problem with this treatment. The program is blatantly written to compute only in directions toward the point Out. For example, the defining equation for transistor T1 does not connect its d-input. The diagram to the right of the program shows the directionality imposed on CKT in this way. Only with transistor T6 are both directions used.

If transistor T1 is open (not conducting), and the rest are closed, then according to the definitions for NT, PT and DOTs, Out is

$$
\begin{aligned}
\texttt{CKT [H, H, H, H, H, H]} &= \texttt{T1d} \cdot \texttt{T2d} \cdot \texttt{T3d} && \text{(eqn. for Out)} \\
&= \texttt{Z} \cdot \texttt{T2d} \cdot \texttt{T3d} && \text{(T1 is open)} \\
&= \texttt{T2d} \cdot \texttt{T3d} && (\texttt{Z} \cdot v = v) \\
&= (\texttt{T4d} \cdot \texttt{T6d}) \cdot \texttt{T3d} && \text{(T2 is closed)} \\
&= \texttt{L} \cdot \texttt{T6d} \cdot \texttt{3d} && \text{(T4 is closed)} \\
&= \texttt{L} \cdot \texttt{T5d} \cdot \texttt{3d} && \text{(T6 is closed)} \\
&= \texttt{L} \cdot \texttt{L} \cdot \texttt{T3d} && \text{(T5 is closed)} \\
&= \texttt{L} \cdot \texttt{T3d} && (\texttt{L} \cdot \texttt{L} = \texttt{L}) \\
&= \texttt{L} \cdot \texttt{T5d} \cdot \texttt{T6s} && \text{(T3 is closed)} \\
&= \texttt{L} \cdot \texttt{L} \cdot \texttt{T6s} && \text{(T5 is closed)} \\
&= \texttt{L} \cdot \texttt{T6s} && (\texttt{L} \cdot \texttt{L} = \texttt{L}) \\
&= \texttt{L} \cdot \texttt{T4d} && \text{(T6 is closed)} \\
&= \texttt{L} \cdot \texttt{L} && \text{(T4 is closed)} \\
&= \texttt{L} && \text{(defn. DOTs)}
\end{aligned}
$$

Since these programs are in a lazy language, this derivation is essentially a trace

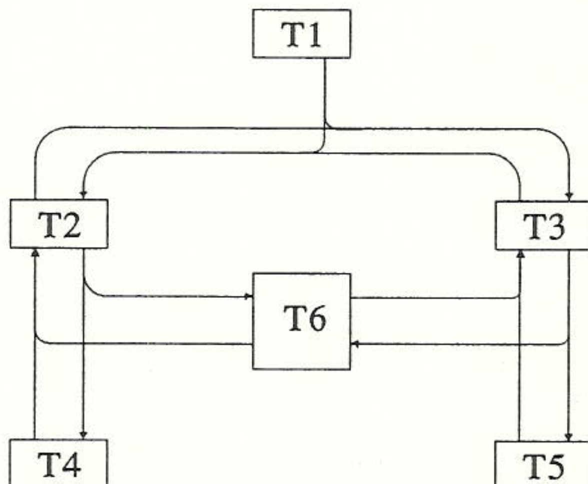
of the actual computation. Expressing more of the connectivity would lead to trouble in this programming style. For example, if the equations for T2, T3, and T6 were changed to

```
              ⋮
[T2s, T2d] = NT [g2, DOT [T6d, T4d], DOT [T3d, T1d]]
[T3s, T3d] = NT [g3, DOT [T6s, T5d], DOT [T2d, T1d]]
[T6s, T6d] = NT [g6, DOT [T3s, T5d], DOT [T2s, T4d]]
              ⋮
```

the derivation would be

$$
\begin{array}{lll}
\texttt{CKT [H, H, H, H, H, H]} & = \texttt{T1d}\cdot\texttt{T2d}\cdot\texttt{T3d} & (\text{eqn. for } \texttt{Out}) \\
& = \texttt{Z}\cdot\texttt{T2d}\cdot\texttt{T3d} & (\texttt{T1} \text{ is open}) \\
& = \texttt{T2d}\cdot\texttt{T3d} & (\texttt{Z}\cdot v = v) \\
& = (\texttt{T6d}\cdot\texttt{T4d})\cdot\texttt{T3d} & (\texttt{T2} \text{ is closed}) \\
& = (\texttt{T3s}\cdot\texttt{T5d})\cdot\texttt{T4d}\cdots & (\texttt{T6} \text{ is closed}) \\
& = (\texttt{T2d}\cdot\texttt{T1d})\cdot\texttt{T5d}\cdots & (\texttt{T3} \text{ is closed}) \\
& = (\texttt{T6d}\cdot\texttt{T4d})\cdot\texttt{T1d}\cdots & (\texttt{T2} \text{ is closed}) \\
& \vdots &
\end{array}
$$

A leftmost derivation diverges because the values of certain transistor outputs depend on themselves. In this example, there is a cyclic dependence along the path (T2, T6, T3, T2, ...). The other cycles are shown in the diagram



A global analysis is needed to avoid coding cyclic dependence. It is based on what points in the circuit are to be regarded as outputs. A different instance of the CKT program would be needed to compute the voltage at the point I3. Solutions to this problem are developed in Sections 4 and 5. However, before looking at these, let us briefly consider a different symbolic interpretation of CKT.

3.1. Integrating with Verification

The `CKT` expression can be interpreted as a predicate, and the `CKT` program can synthesize a formula for that interpretation. The examples below simply illustrate the flexibility of symbolic processing, showing how functional modeling might integrate with other systems for verification or synthesis.

O'Donnell demonstrates with HDRE how redefinitions of the ground type permit a single circuit description to model physical and geometric aspects of an implementation [16]. In a slightly different approach, Boute parameterizes a semantics of an applicative metalanguage to accommodate various models of circuitry [2].

The primitives for building expressions or their representations are not presented here. Like most Lisp dialects, Daisy has a homogeneous representation for programs and data, making it easy to build expressions out of expression fragments. Since Daisy is interpreted, expressions thus built can be directly evaluated.

Only the functions `PWR`, `GND`, `DOT`, `NT`, and `PT` are redefined to build formulas as indicated below.

Expression	*builds the formula*
`PWR`	$\mathbf{P}$
`GND`	$\mathbf{G}$
`DOT` [A, B]	$A \wedge B$
`NT` [G, S]	$G \supset S$
`PT` [G, S]	$\neg G \supset S$

With these revised definitions, and the original definition of `CKT`, the expression `CKT ["L", "L", "H", "H", "H", "L"]` builds

$$
\begin{aligned}
&(\neg L \supset \mathbf{P})\\
&\wedge [L \supset ((H \supset \mathbf{G}) \wedge (L \supset (H \supset \mathbf{G}))]\\
&\wedge [H \supset ((H \supset \mathbf{G}) \wedge (L \supset (H \supset \mathbf{G}))]
\end{aligned}
$$

Interpreting `H` as `TRUE`, `L` as `FALSE`; and leaving $\mathbf{P}$ and $\mathbf{G}$ as free variables; this expression simplifies to $\mathbf{P} \wedge \mathbf{G}$. Extending logical connectives to deal with the four-valued voltage basis, this formula says that the circuit is shorted.

Similarly, the expression `CKT ["g1", "g2", "g3", "g4", "g5", "g6"]` returns the formula

$$
\begin{aligned}
&(\neg g_1 \supset \mathbf{P})\\
&\wedge [g_2 \supset ((g_4 \supset \mathbf{G}) \wedge (g_6 \supset (g_5 \supset \mathbf{G}))]\\
&\wedge [g_3 \supset ((g_5 \supset \mathbf{G}) \wedge (g_6 \supset (g_4 \supset \mathbf{G}))]
\end{aligned}
$$

Call this assertion Q. From $Q \equiv \mathbf{P} \oplus \mathbf{G}$ ($\oplus$ denoting *exclusive-or*), one could deduce the electrical validity condition mentioned in Section 1.1:

$$g_1 \equiv (g_2 \wedge g_4) \vee (g_2 \wedge g_6 \wedge g_5) \vee (g_3 \wedge g_6 \wedge g_4) \vee (g_5 \wedge g_3).$$

4. Indeterminate Directionality

In the previous section, CKT's descriptions were valid only in predetermined input/output orientations. The partial solution in this section models a closed transistor as choosing the value of *some* neighboring voltage source. Each transistor dynamically determines a direction for information flow. The guarded-choice construct, defined and discussed in Section 2, is used to implement this behavior.

```
def NT [g, s, d] =
  if  L?(g) then "Z"
  else
        GIF { GUARD[ OR[ H?(s), L?(s), X?(s)], s]
              GUARD[ OR[ H?(d), L?(d), X?(d)], d]
              GUARD[ Z?(s), d]
              GUARD[ Z?(d), s]
             }
```

The definition for PT is identical except for its test on g. A closed transistor assumes the value of either its "source" input s or its "drain" input d. If s is not connected, the transistor commits to d's value and *vice versa*. If two or more of the GUARD expressions are satisfied, a choice is made. On the other hand, if s can't be determined but d can, NT's result is based on d's value.

A weaker version of DOTs, called DOTw, returns the first connected value.

```
         |
         | DOTw: [ Value ... Value] --> Value
         |
     def DOTw [] = "Z"
         DOTw ["Z" ! Vs] = DOTw(Vs)
         DOTw [ V ! Vs] = V
```

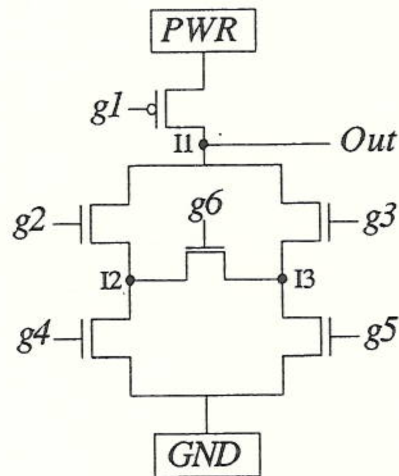
Under these definitions, an expression for CKT can state all the physical connections. For example, the definition below states that transistor T3 is connected

to T1, T2, T6, T5, *and* T3.

```
def CKT [g1, g2, g3, g4, g5, g6] = Out
  where
    Out = I1

     I1 = DOTw {T1, T2, T3}
     I2 = DOTw {T2, T6, T4}
     I3 = DOTw {T3, T6, T5}

     T1 = PT [g1, PWR,  I1]
     T2 = NT [g2,  I1,  I2]
     T3 = NT [g3,  I1,  I3]
     T4 = NT [g4,  I2, GND]
     T5 = NT [g5,  I3, GND]
     T6 = NT [g6,  I2,  I3]
```



The description is pleasantly direct, considering that it is in an applicative source language. Applied to a multiset, DOTw chooses the first voltage it can determine. If DOTs were used here, or if DOTw were applied to a determinate list, this program would fail more often. As it stands, this version of CKT correctly models the circuit's behavior where the gate inputs satisfy the validity condition,

$$g_1 \equiv (g_2 \wedge g_4) \vee (g_2 \wedge g_6 \wedge g_5) \vee (g_3 \wedge g_6 \wedge g_4) \vee (g_5 \wedge g_3).$$

Should the inputs not meet this condition, one of two things happens. When Out is shorted the result is either H or L, depending on the effort needed to find a voltage source. In this case, it is easier to find an H because there are fewer transistors between Out and PWR than between Out and GND.

When Out is unconnected, CKT *may* diverge because DOTw ignores Zs. This can reduce the computation to the kind of cyclic dependence discussed in Section 3. The table below shows the values computed for all H/L combinations of the gates. The question marks show where the program diverges [NOTE 4] and the {H} entries show where CKT fails to discover Xs. For reasons just discussed, there happen to be no "{L}" entries.

Out = I1	← g4,g5,g6 →							
g1,g2,g3	LLL	LLH	LHL	LHH	HLL	HLH	HHL	HHH
LLL	H	H	H	H	H	H	H	H
LLH	H	H	{H}	{H}	H	{H}	{H}	{H}
LHL	H	H	H	{H}	{H}	{H}	{H}	{H}
LHH	H	H	{H}	{H}	{H}	{H}	{H}	{H}
HLL	Z	Z	Z	Z	Z	Z	Z	Z
HLH	?	?	L	L	?	L	L	L
HHL	?	?	?	L	L	L	L	L
HHH	?	?	L	L	L	L	L	L

Despite its problems, this treatment permits us to describe CKT in a compositional fashion, without regard to the surrounding directionality. For example, if "Out = I1" is replaced by "Out = I2," and with no other changes, the program computes the values below for the junction of transistors T2, T4, and T6.

Out = I2	←			g4,g5,g6				→
g1,g2,g3	LLL	LLH	LHL	LHH	HLL	HLH	HHL	HHH
LLL	Z	?	Z	L	L	L	L	L
LLH	Z	H	Z	L	L	L	L	L
LHL	H	H	H	H	L	L	L	L
LHH	H	H	H	L	L	L	L	L
HLL	Z	?	Z	L	L	L	L	L
HLH	Z	?	Z	L	L	L	L	L
HHL	?	?	?	L	L	L	L	L
HHH	?	?	L	L	L	L	L	L

The curious reader might determine which of these entries are undetected shorts. The not-so-curious may consult the table at the end of the next section.

5. Bidirectional Modeling

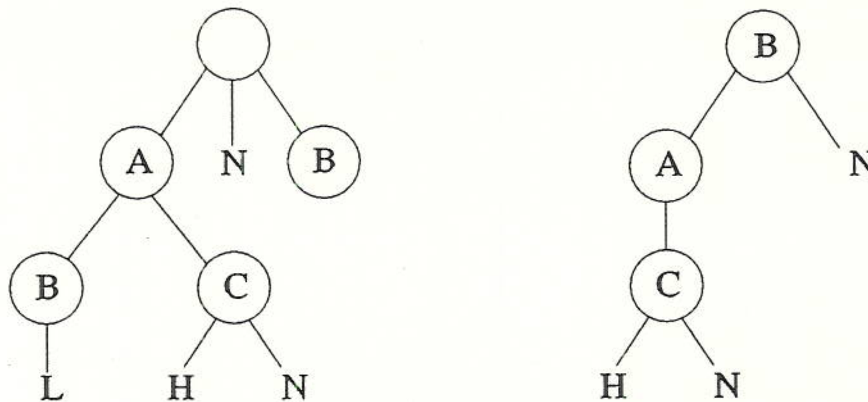
In order to detect shorts and no-connects, each transistor must examine all its neighbors and avoid self reference while doing so. This is accomplished by giving each transistor a distinct name. We develop the following data structures. List representations for and operations on these structures, all quite ordinary, are shown in Appendix A.

A *labeled join* is a tree whose leaves are values and whose interior nodes are labeled. An *(unlabeled) join* is just like a labeled join, except that its root is not labeled. The operations on joins (See Appendix A) are

```
| LABELj : Label --> Join --> LabeledJoin
|  JOINj : [LabeledJoin ... LabeledJoin] --> Join
| MERGEj : [Join, Join] --> Join
|   NEWj : [Label, Value] --> LabeledJoin
|   DOTj : Join --> Value
```

LABELj places a label at the root of a join and removes all subtrees with that label. The figure below, on the right, shows the result of applying LABELj("B")

to the unlabeled join on the left.



NEWj creates a new labeled join from a label and value. MERGEj "connects" two unlabeled joins by superimposing their roots. JOINj places several joins under a new root, creating a new unlabeled join. DOTj evaluates a join by forming the product of its leaves. Like DOTs, DOTj terminates when an X is discovered. DOTj returns X when applied to the left tree and H when applied to the right tree.

An N-transistor is modeled as follows. As usual, a P-transistor uses the L? test in place of H?. Once again, gates are treated as pure inputs.

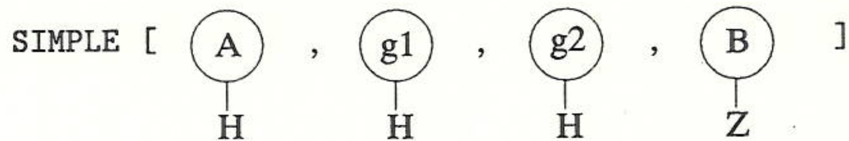
```
     |
     | NT: Label --> [Join, Join, Join] --> LabeledJoin
     |
def NT(Me) = λ[g, s, d].
  if H?(DOTj(g))
     then  (LABELj(Me)) (MERGEj [s, d])
     else  NEWj [Me, "Z"].
```

The parameter Me is this transistor's unique label. The transistor evaluates the voltage at its gate. If the gate is true, the result is a labeled join connecting the transistor's source and drain inputs with a root labeled Me. If the gate is false, a labeled join representing no connection is returned.

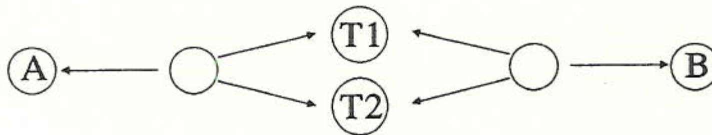
LABELj recursively cancels self references. To see how this happens, let us look at a simple combination of transistors [NOTE 5].

```
def SIMPLE [A, g1, g2, B] = [A', B']
   where
     T1 = (NT "T1") [g1, A', B']
     T2 = (NT "T2") [g2, A', B']
     A' = JOINj [A, T1, T2]
     B' = JOINj [B, T1, T2]
```

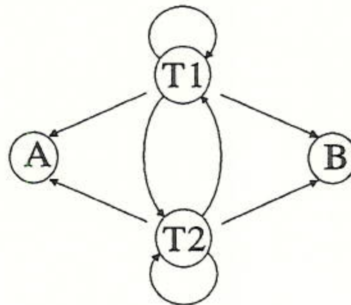
The arguments to SIMPLE are labeled joins; for instance, we might evaluate



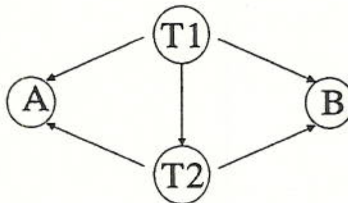
SIMPLE produces outputs A' and B' for its external ports, A and B. These are unlabeled joins of the external inputs and the two transistors.



In the case that both gates are false, each transistor simultaneously combines A' and B'. The resulting graph would have the cyclic structure



However, LABELj(Me) eliminates paths from any transistor to itself. Since this is done simultaneously by all the transistors, all cycles are cut. The diagram below shows only T1's view of the result, a finite acyclic graph representing all paths through the circuit.



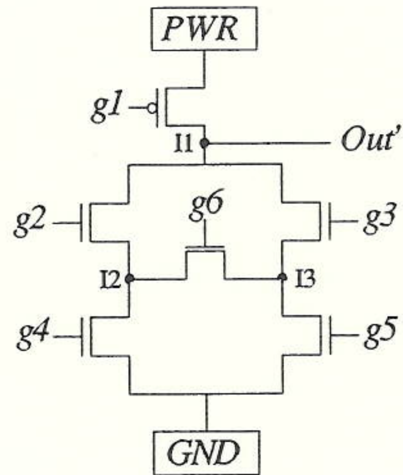
Hence, evaluation of T1, T2, A', or B' always terminates.

The CKT example is expressed as

```
def CKT [Out, g1, g2, g3, g4, g5, g6] = Out'
  where
   Out' = I1

     I1 = JOINj [Out, T1, T2, T3]
     I2 = JOINj [ T2, T6, T4]
     I3 = JOINj [ T3, T6, T5]

     T1 = (PT "T1")[g1, PWR, I1]
     T2 = (NT "T2")[g2,  I1, I2]
     T3 = (NT "T3")[g3,  I1, I3]
     T4 = (NT "T4")[g4, GND, I2]
     T5 = (NT "T5")[g5, GND, I3]
     T6 = (NT "T6")[g6,  I2, I3]
```



This expression is bidirectional. It requires a contributing voltage from the port Out. With Out unconnected, this program computes the the same table of values for point Out' as is shown in Section 3. The value of I2, joining transistors T2, T4, and T6, is computed by the same expression, replacing Out' = I1 with Out' = I2.

Out' = I2	← g_4, g_5, g_6 →							
g_1, g_2, g_3	LLL	LLH	LHL	LHH	HLL	HLH	HHL	HHH
LLL	Z	Z	Z	L	L	L	L	L
LLH	Z	H	Z	X	L	X	L	X
LHL	H	H	H	X	X	X	X	X
LHH	H	H	X	X	X	X	X	X
HLL	Z	Z	Z	L	L	L	L	L
HLH	Z	Z	Z	L	L	L	L	L
HHL	Z	Z	Z	L	L	L	L	L
HHH	Z	Z	L	L	L	L	L	L

Thus, this program models bidirectional behavior at the intended level of detail. It solves simultaneous systems of transistor equations over a finite set of voltage values. Each transistor builds a data representation of the entire circuit, from its own vantage point. Since these structures are built recursively, they are cyclic graphs. However, by editing its own *join* structures a transistor removes infinite paths involving itself. This is done uniformly by all the transistors, resulting in a tree of connected voltages for each. Evaluation of these trees by DOTj delivers values for the junctions. The treatment is compositional in the sense that the work is done by the programs PT and NT, and not by a global analysis for directionality.

6. Remarks, Directions, and Conclusions

We have explored various symbolic representations for transistors in a four-valued domain of behavior. The modeling was done in an applicative surface language over a general purpose graph processing system. Because the language is lazy, the structure of the circuit can be expressed directly as a recursive data dependence.

The examples focus on the treatment of bidirectionality. Being bidirectional does not necessarily mean being relational. In digital design, most transistors have a logical orientation. Those that do not often develop a direction from the circuit's external environment. That is, every conductor has a direction at any time, although it may change from time to time.

One can describe bidirection applicatively by naming the two directions of information flow, as is done in Section 3. At a notational level, the same kind of thing is done by Musser, Narendan, and Premerlani in BIDS [15]. Doing this uniformly is a bit cumbersome, and a global orientation was needed anyway to avoid divergent recursions. Divergence is not a problem for BIDS, which verifies assertions about behavior.

In Section 3.1, the base functions are altered to synthesize propositional formulas directly from a directional version of `CKT`. This is simply a matter of using the available symbolic processing facilities. This program could not produce the *quantified formulas* used in predicate descriptions of VLSI (e.g. [7]). This is a programming challenge, involving techniques used in Sections 4 and 5.

In Section 4, the need for a global analysis is eliminated by coding versions of the transistors which dynamically choose a direction. This is closer to a relational style of description, but the programs in Section 4 fail to detect shorts and sometimes diverge. Ambiguous constructs such as Daisy's multiset expressions raise problems in the semantics of functional languages. They weaken their very functionality, complicating not only implementation, but also the rules of reasoning between expressions and their values. In programming, these constructs are typically used for their operational qualities; Section 4 is an example. Daisy's `{ ··· }` constructor is not nondeterministic, which is not to say that it is deterministic. It is merely concurrent. The `GIF` conditional avoids divergence but does not make the best (or worst) choice of evaluation order. However, because these constructs lend a quasi-relational quality to applicative notation, they are worth investigating in this context.

The final treatment in Section 5 implements deadlock avoidance by giving each transistor a name. The `CKT` program is highly circular but the cycles are systematically eliminated. It is hard to imagine adopting this representation in an applicative order modeling language. In the third chapter of their text book, Abelson and Sussman develop a contrasting approach to electrical modeling [1].

This paper reports experimentation toward a symbolic modeling methodology. The programs can certainly be improved, through a combination of the techniques illustrated and general program improvement. The following comments refer to the programs in Section 5.

- Treating gates as pure inputs gains brevity at a cost in generality. It is a minor refinement to model a transistor as providing an output for its gate as well as its source and drain. A fully bidirectional inverter description might then look like

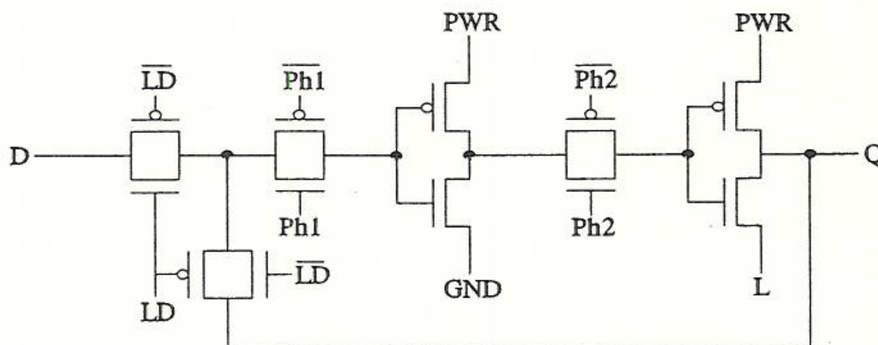
```
INV [In, Out] = [In', Out']
  where
    [T1g, T1s, T1d] = (PT "T1") [ In', Out',  PWR ]
    [T2g, T2s, T2d] = (NT "T2") [ In',  GND, Out' ]
                In' = JOINj [ In, T1g, T2g]
               Out' = JOINj [Out, T1d, T2d]
```

It can be demonstrated that, while transistors can be turned around, the inverter works in only one direction.

- Each transistor must have a unique reference designation. In Section 5 this was done in by using literals "T1", "T2", and so on. Hierarchical descriptions must somehow ensure this uniqueness. It could be done by parameterizing each level of description; for example, `CKT` would concatenate its own `Me` parameter to the designations within. Of course, `Me` doesn't have to be a literal, nor even printable. In some examples not presented here, we shamelessly exploited the list allocator by letting each join serve as its own label.

- Though the transistor expressions in Section 5 share a lot of information, they do not share much structure. Each one builds a distinct representation for its view of the system. Consequently, these programs do a lot of redundant computation. The version of `CKT` in Section 4 does not have this problem; partial results are shared.

- These modeling techniques apply to circuits with state. As one would expect, notions of strength and capacitance (or at least delay) are needed. With these refinements to the base programs of Section 5, the behavior of such circuits as this D-type flip flop [19, Fig. 5.44a] can be modeled.



Further details can be found in [3]. The enhancements bring us closer to the level of description addressed by Winskel, Musser, and others, cited in Section 1.1.

In summary, there is every prospect that symbolic modeling will play an important role in VLSI engineering. Hence, part of the effort in formal hardware research must be to show how *its* tools can be used in practice. We are only beginning to develop methodology for animating hardware descriptions. Our purpose has been to illustrate how facets of design can be isolated. The models shown here are useful for limited explorations of VLSI behavior. They are also disposable because little effort is involved in building them.

Notes

1. The variable `d` is included for correspondence to the schematic. It names the output. It could be eliminated from the definitions by writing

   ```
   def NT [g, s] = if H?(g) then s else "Z"

   def PT [g, s] = if L?(g) then s else "Z"
   ```

   The `d` is used later to identify applied occurrences of `NT` and `PT`. For example, the `INV` program uses `T1d` for the drain of transistor `T1`. In later programs it is more useful to have such names.

2. `INV` could be defined by a single term,

   ```
   def INV(i) = DOT [ PT [i, PWR], NT [i, GND] ]
   ```

3. The expression for `I1` could be

   ```
   DOTs [T1d, T2d, T3d]
   ```

   It is written this way because this version of `CKT` is used for a different purpose in Section 1.1.

4. For this table and the next one, `CKT` was run for a period of time judged sufficient to generate an answer if one existed. Of course, this is hardly a test for divergence; however, by looking at the configuration of gates for '?' entries, divergence can be determined. For example, `CKT [H, H, L, L, L, L]` closes just transistor `T2`. Symbolic evaluation of the equations shows that in this case, the equation for `Out` becomes

   ```
   Out = DOTw {T1, T2, T3}
       = DOTw {"N", Out, "N"}
       = Out
   ```

   Hence, the program does diverge in this case.

5. Think of T1 and T2 as transmission gates, ignoring the fact that N-type transistors don't pass high voltage very well.

References

[1] Abelson, Harold and Sussman, Gerald Jay with Sussman, Julie, *Structure and Interpretation of Computer Programs,*, (The MIT Press, Cambridge, 1985).

[2] Boute, Raymond T., System semantics and formal circuit description, *IEEE Transactions on Circuits and Systems* **CAS-33** (12) (December,. 1986) 1219–1231

[3] Boyer, C. David and Johnson, Steven D., Modeling transistors with Daisy, Indiana University Computer Science Department Technical Report,. in progress

[4] Bryant, Randell E., A switch-level model and simulator for MOS circuits, *IEEE Transactions on Computers* **C-33**, No. 2 (February, 1984), 160–177.

[5] Dhingra, I. S., Formal validation of an integrated circuit design style, in Birtwistle, Graham and Subrahmanyam, P. A. (eds.), *VLSI Specification, Verification and Synthesis,* (Kluwer, Boston, 1988).

[6] Gopalakrishnan, Ganesh C., Smith, D. R., and Srivas, M. K., An algebraic approach to the specification and realization of VLSI designs, *Proc. 7th Symposium on Computer Hardware Description Languages,*Tokyo, Japan (North-Holland, Amsterdam, 1985).

[7] Gordon, M. J. C., Why higher-order logic is a good formalism for specifying and verifying hardware, , in Subramanyam, P. A. and Milne, G. J. (eds.), *Formal Aspects of VLSI design,* (North-Holland, Amsterdam, 1986), 153–178.

[8] Hanna, F. K. and Daeche, N., Specification and verification using higher-order logic: a case study, in Subramanyam, P. A. and Milne, G. J. (eds.), *Formal Aspects of VLSI design,* (North-Holland, Amsterdam, 1986), 179–213.

[9] Henderson, Peter, *Functional Programming—Application and Implementation,* (Prentice-Hall, Englewood Cliffs, 1980).

[10] Henson, Martin C., *Elements of Functional Languages,*, (Blackwell Scientific Publications, Oxford, 1987).

[11] Johnson, Steven D., *Synthesis of Digital Designs from Recursion Equations,*, (The MIT Press, Cambridge, 1984).

[12] Johnson, Steven D., Daisy Programming Manual, Draft manual, available on request, Indiana University Computer Science Department, Bloomington, Indiana.

[13] Mehlam, Thomas F., Abstraction Mechanisms for hardware verification, in Birtwistle, Graham and Subrahmanyam, P. A. (eds.), *VLSI Specification, Verification and Synthesis,* (Kluwer, Boston, 1988) 217–233.

[14] Milne, George J., CIRCAL and the representation of communication, concurrency, and time, *ACM Trans. on Programing Languages and Systems,* 7 2 (April, 1985) 270–298.

[15] Musser, David R., Nerendran, Pakiath , and Premerlani, William J., BIDS: A method for specifying and verifying bidirectional hardware devices, in Birtwistle, Graham and Subrahmanyam, P. A. (eds.), *VLSI Specification, Verification and Synthesis,* (Kluwer, Boston, 1988) 217–233.

[16] O'Donnell, John T., Hardware description with recursion equations, *Proc. 8th International Symposium on Computer Hardware Description Languages and their Applications,* Amsterdam, April, 1987..

[17] Peyton-Jones, Simon L., *The Implementation of Functional Programming Languages,*, (Prentice Hall, Englewood Cliffs, 1987).

[18] Sheeran, Mary, μFP, a language for VLSI design, *Conf. Rec. 1984 Symp. on LISP and Functional Programming, 104–112.*

[19] Weste, Neil and Eshraghian, Kamran, *Principles of CMOS VLSI Design, A Systems Perspective,*, (Addison-Wesley, Reading, 1985).

[20] Winskel, Glynn, Models and logic of MOS circuits, in Birtwistle, Graham and Subrahmanyam, P. A. (eds.), *VLSI Specification, Verification and Synthesis,* (Kluwer, Boston, 1988).

Appendix A. Representation of Joins

These programs represent the *join* objects used in Section 5.

```
|
|   LABELj : Label --> Join --> LabeledJoin
|   ELIM   : [Join ... Join]  --> [Join ... Join]
|
def LABELj (Me) = λ Js. [Me ! ELIM(Js)]
  where
    ELIM (Js) =
       let [[L ! J0] ! J1] = Js in
         if nil?(isLST? Js) then Js else
         if same? [Me, L] then ELIM (J1)
         else [[L ! ELIM (J0)] ! ELIM(J1)]
```

```
|
|  JOINj : [LabeledJoin ... LabeledJoin] --> Join
|
def JOINj(Js) = Js
|
| MERGEj : [Join, Join] --> Join
|
def MERGEj = APPEND
|
|   NEWj : [Label, Value] --> LabeledJoin
|
def NEWj [L, Vs] = [[L ! Vs]]
|
|   DOTj : Join --> Value
|   FLAT : [Join, [Value ... Value]] --> [Value ... Value]
|
def DOTj(J) = DOTs(FLAT [J, []])
  where
    FLAT [J, Etc] =
      let [[L ! V] ! J'] = J
       in
      let Etc' = FLAT[J', Etc]
       in
          if nil?(J) then Etc else
          if nil?(V) then Etc' else
          if not?(list?(V)) then [V ! Etc']
          else FLAT[V, Etc']
```