NASA Contractor Report 191509

A Formal Language for the Specification and Verification of Synchronous and Asynchronous Circuits

David M. Russinoff

COMPUTATIONAL LOGIC, INC.
Austin, Texas

Contract NAS1-18878
September 1993

NASA Contractor Report 191509

A Formal Language for the Specification and Verification of Synchronous and Asynchronous Circuits

David M. Russinoff

Computational Logic, Inc.
1717 West Sixth Street, Suite 290
Austin, Texas 78703

Contract NAS1-18878 September 1993

1 Introduction

Our past work in the formalization and verification of fault-tolerant systems has consisted of three tasks:

1. The formal design and verification of a circuit that achieves Byzantine agreement among four synchronous processors [1];
2. The mechanical verification of the Interactive Convergence clock synchronization algorithm [9];
3. The formalization of the Biphase Mark protocol for asynchronous communication [7].

The purpose of the present task, Task 4, is to investigate the integration of these previous efforts in the design of an asynchronous Byzantine-resilient computing system. The ultimate goal is a formally verified gate-level implementation.

The design of a hardware circuit that achieves asynchronous communication is necessarily contingent on an underlying model of hardware behavior. Any attempt to devise such a circuit in the abstract, without first establishing a suitable model, would be a largely wasted effort. Thus, a prerequisite for the realization of our goal is the selection of a formal hardware description language (HDL), along with an underlying behavioral model.

Our previous research in hardware modeling and verification has been based on an HDL developed at CLI by Brock and Hunt [5]. The utility of the Brock-Hunt HDL as a verification tool, as demonstrated in the verification of the FM9001 microprocessor [4], stems from the simplicity of its semantics. All circuits designed in this language are assumed to be driven by an implicit global clock. Simulation of a circuit amounts to a computation of a sequence of states corresponding to clock cycles. Thus, no explicit representation of time or propagation delays is provided, so that the class of circuits that can be satisfactorily modeled is limited. In particular, the language is unsuitable for any application involving asynchrony.

Commercial event-driven simulation languages provide for a broader range of hardware behaviors. VHDL [7] in particular has gained wide acceptance in the hardware design community as a validation tool. Since the limitations of simulation as a method of validation are well known, a formal verification

system based on VHDL would have clear practical value. Unfortunately, like most programming languages in common use, VHDL was not intended as an object of reason. Inevitably, its semantics are complicated and obscure. Various attempts to formalize VHDL [2,6] have encountered severe difficulty and show limited promise of short-term success.

We have undertaken, therefore, to develop a new formal HDL with the intended application of verifiable asynchronous communication. This paper is a report on the progress of this endeavor. Our primary objective is to formalize the event-based behavioral model of VHDL while retaining the semantic clarity of the Brock-Hunt HDL. Thus, we would like to inherit the proof methodology developed by Brock and Hunt, including

- abstract descriptions of (acyclic) combinational circuits in terms of Boolean functions;
- abstract state machine descriptions of sequential devices;
- hierarchical design and verification of complex circuits.

At the same time, our language should provide for

- faithful implementation of the VHDL notions of time and propagation delay;
- gate-level construction of sequential devices by means of feedback loops, e.g., flip-flops implemented by cross-coupled nand gates;
- modeling of asynchronous communication.

Following [5], we have developed our language within the logical framework of the Nqthm system of Boyer and Moore [3]. Its simulator (operational) semantics are expressed by a recursive function `SIM`, defined in the Nqthm logic. The two principal arguments of this function are a hardware module to be simulated and a list of *waveforms* corresponding to the module's input signals, representing the values of those signals as functions of time. The value returned by `SIM` is a list of waveforms corresponding to the module's output signals, produced by propagating the input values according to the structure of the module.

In Section 2, we describe our formal notion of time, the structure of waveforms, and the propagation of signal values. In addition to the two

conventional VHDL delay modes, *transport* and *inertial*, we define a *nondeterministic* mode, which subsumes the other two and provides a scheme for concise behavioral descriptions of combinational circuits. We also introduce the notion of an *indefinite* or partially specified waveform, which is critical to the subsequent development as it provides for the simulation of abstract modules with partially defined behaviors.

In Section 3, we discuss our representation of hardware modules as Nqthm constants. The behaviors of combinational, sequential, and structural circuits are defined by means of a `STEP` function, for which we provide an axiomatic characterization. The top-level simulator function `SIM` is defined recursively in terms of `STEP`, as described in Section 4. The simulator is complicated considerably by the possible presence of *delta* delays, which represent zero-delay devices as prescribed in the VHDL standard [7]. In conformance with commercial VHDL simulators, in order to guarantee that simulation terminates, an extra argument is passed to `SIM`, representing a uniform bound on the lengths of all zero-delay paths within a circuit. Related constraints are also imposed on the input waveforms to a module.

Our approach to circuit verification is based on formal notions of module *specification* and *implementation*, as presented in Section 5. Here, we also describe procedures that automatically derive abstract combinational and sequential behavioral modules that serve as specifications for circuits of certain types. Once a behavioral module has been proved to be a specification of a circuit, it may be substituted for any instances of the circuit that occur as components of any larger circuit without affecting its functionality. This principle is the key to hierarchical circuit analysis.

Included in the text are informal statements of some general theorems that are relevant to the verification of circuits defined in our system. The proofs of most of these theorems remain to be checked mechanically, and this work will be a significant portion of the Task 5 effort. The function definitions that compose the simulator, on the other hand, have all been formally accepted by the Nqthm prover, and appear in complete form in an appendix.

2 Signals

2.1 Time

We define a *time* to be an ordered pair of natural numbers, as recognized by the predicate `TIMEP` (see the appendix for the definition of this and all other functions referenced herein). The set of all such pairs is ordered lexicographically. Thus, the time origin, the least element of this set, is the pair `'(0 . 0)`.

The first component of a time represents the number of time units, which we arbitrarily take to be picoseconds, that have elapsed since the start of a simulation. The second component, which we call the *delta* component, is required in order to allow zero-delay events. It represents the number of successive zero-delay events that have been scheduled during the current time unit.

The time for which an event is scheduled is computed from the current time `t0` and a given propagation delay by the function `TPLUS`. If the delay is 0, then the value returned is the result of incrementing the second component of `t0` by 1; otherwise the delay is added to the first component of `t0` and the second component is set to 0.

2.2 Waveforms

A *waveform* is a function that assigns a value to every time. In our formalization, we represent a waveform as an association list. Each pair in this list consists of a value and a time at which that value was or is to be assumed by the signal with which the waveform is associated. These pairs, which are called *events*, are listed in decreasing order with respect to time. The time of the earliest event in any waveform is `'(0 . 0)`.

There is no restriction on the values that may be assumed by a signal. We adopt the convention of using the the symbols `'T` and `'F` to represent high and low signal values, respectively. The value `'X` is special—it represents an unknown value. Any value other than `'X` is said to be *definite*. A *definite waveform* is one that never assumes the value `'X`. A value `v1` *generalizes* a value `v2` if either `v1` = `v2` or `v1` = `'X`. A waveform `w1` *generalizes* a waveform `w2` if for every time `t`, the value of `w1` at `t` generalizes the value of `w2` at `t`. Note that the set of all waveforms is a lower semi-lattice under this relation. This

means that any two waveforms have a greatest lower bound, i.e., a common generalization that is generalized by any other common generalization. The set also has a least element with respect to this ordering, which we call the *null waveform*, namely the waveform `'((X . (0 . 0)))`, which assigns the value `'X` to every time.

A list of waveforms is called a *packet*. The lattice structure is extended in the obvious manner to the set of all packets of any fixed length. Thus, one packet is said to generalize another if the relation holds between corresponding waveforms.

2.3 Propagation

The functions `POST-INERTIAL-EVENT-DEFINITE` and `POST-TRANSPORT-EVENT-DEFINITE` implement *inertial* and *transport* delay, as defined in the VHDL standard[5]. Each of these functions takes as arguments a waveform `w`, a value `v`, and a time `t1` at which `v` is to be scheduled on `w`. `POST-INERTIAL-EVENT-DEFINITE` takes as an additional argument the current time `t0`, which must precede `t1`. (The effect of scheduling an event with transport delay is independent of the current time.) The value returned by either function is the appropriately modified waveform.

However, the correctness of these functions depends on the assumption that both `v` and `w` are definite. If we allow either argument to be indefinite, then the more general functions `POST-INERTIAL-EVENT` and `POST-TRANSPORT-EVENT` must be used. While the recursive definitions of these functions are quite complicated, they may be described informally but precisely as follows:

> For any waveform `w`, any `v`, and any times `t0` $<$ `t1` $\leq$ `t2`, the value of `(POST-INERTIAL-EVENT w v t0 t1 t2)` is the greatest lower bound of the set of all waveforms of the form `(POST-INERTIAL-EVENT-DEFINITE w' v' t0 t')`, where `w'` is generalized by `w`, `v'` is generalized by `v`, and `t1` $\leq$ `t'` $\leq$ `t1`.

> For any waveform `w`, any `v`, and any times `t0` $<$ `t1` $\leq$ `t2`, the value of `(POST-TRANSPORT-EVENT w v t0 t1 t2)` is the greatest lower bound of the set of all waveforms of the form `(POST-TRANSPORT-EVENT-DEFINITE w' v' t')`, where `w'` is generalized by `w`, `v'` is generalized by `v`, and `t1` $\leq$ `t'` $\leq$ `t1`.

These are the two functions that are actually called by our simulator to schedule events for signals with inertial and transport delay, respectively. Note that in addition to accepting an indefinite value and an indefinite initial waveform, they also accept a range of possible times instead of a definite time for the scheduling of the event. In order to schedule an event for a definite time `t`, the appropriate function is called with `t2` = `t1` = `t`.

We also introduce a third mode of propagation delay, called *nondeterministic delay*, which is implemented by the function `POST-NONDETERMINISTIC-EVENT`. The behavior of this function may be described as follows:

> Given any waveform `w`, any `v`, and any times `t0` $<$ `t1` $\leq$ `t2`, let `tmin` be the minimum of `t1` and the times of any events scheduled on `w` after `t0`. `(POST-NONDETERMINISTIC-EVENT w v t0 t1 t2)` is the waveform whose value at any time `t` is
>
> (a) the value of `w` at `t`, if `t` $<$ `tmin`;
>
> (b) `'X`, if `tmin` $\leq$ `t` $<$ `t2`;
>
> (c) `v`, if `t2` $\leq$ `t`.

This delay mode is not actually exhibited by any primitive devices, but turns out to be useful in the behavioral specification of complex circuits. Its utility, as we shall see in Section 5, stems from the observation that it subsumes both inertial and transport modes in the sense that the waveform `(POST-NONDETERMINISTIC-EVENT w v t0 t1 t2)` is a generalization of both `(POST-INERTIAL-EVENT w v t0 t1 t2)` and `(POST-TRANSPORT-EVENT w v t0 t1 t2)`.

3 Modules

Our simulator accepts three types of modules: *combinational*, *sequential*, and *structural*. A combinational or sequential module is also called *behavioral*. A structural module represents a circuit constructed from behavioral modules. Associated with a module of any type are a fixed number of inputs and a fixed number of outputs.

We define an *input packet* (resp., *output packet*) for a given module to be a list of waveforms whose length is the number of its inputs (resp., outputs).

The behavior of a behavioral module is characterized by a function `STEP` of four arguments: (1) a module `mod`, (2) an input packet `inp`, whose length is the number of inputs of `mod`, (3) an output packet `outp`, whose length is the number of outputs of `mod`, and (4) a time `t`. The value returned by `STEP` is the result of updating `outp` by executing any events in `inp` that are scheduled for time `t`. This function is defined so as to exhibit the following critical properties:

1. **Monotonic**: if `inp1` and `outp1` generalize `inp2` and `outp2`, respectively, then `(STEP mod inp1 outp1 t)` generalizes `(STEP mod inp2 outp2 t)`.

2. **Nonpredictive**: If `inp1` and `inp2` have the same values at all times that are not later than `t`, then `(STEP mod inp1 outp t)` = `(STEP mod inp2 outp t)`. Thus, the past and projected future behavior of a module is independent of its future input.

3. **Nonretroactive**: The values of the updated packet `(STEP mod inp outp t)` at any time no later than `t` are the same as those of `outp`. Thus, the past behavior of a module is immutable.

3.1 Combinational Modules

The simplest modules are combinational. They consist of four components: (1) a list of symbols representing input signals; (2) a list of output forms, which express the values of the output signals in terms of the values of the input signals; (3) a delay mode corresponding to each output signal; and (4) a delay range, represented as a pair of numbers, corresponding to each output signal. A combinational module is *primitive* if all of its delay modes are `inertial` or `transport` and all of its delay ranges are intervals of length 0. As an example of a primitive module, we define a nand gate as follows:

```
(DEFN NAND ()
  '(COMBINATIONAL     ;type
    (A B)             ;inputs
    ((M-NAND A B))    ;outputs
    (INERTIAL)        ;modes
    ((2000 . 2000)))) ;delays
```

This module has two inputs and a single output with a fixed inertial delay of 2000 picoseconds. (Note that a fixed delay is represented as a degenerate range.)

Output forms must be defined in terms of monotonic functions, in order to conform to the monotonicity requirement for our `STEP` function. Thus, the function `M-NAND`, which is used to compute the output of our nand gate, is defined by

```
(DEFN M-NAND (A B)
  (IF (EQUAL A 'F) 'T
    (IF (EQUAL B 'F) 'T
      (IF (AND (EQUAL A 'T) (EQUAL B 'T)) 'F
        'X))))
```

Monotonic versions of other Boolean functions (`M-NOT`, `M-OR`, etc.) are defined similarly.

Execution of `(STEP mod inp outp t)` for a combinational module `mod` amounts to updating each waveform in the packet `outp` by means of a call to the appropriate event-posting function, using the value computed from the corresponding output form and the current input values at time `t`. Thus, for combinational modules, the nonpredictive property of `STEP` may be strengthened as follows: If `inp1` and `inp2` have the same values *at time* `t`, then `(STEP mod inp1 outp t)` = `(STEP mod inp2 outp t)`.

3.2 Sequential Modules

A sequential module consists of ten components: (1) a list of input signals; (2) a list of output forms; (3) a list of delay modes; (4) a list of delay ranges; (5) a *trigger*, which may be either `POSITIVE-EDGE` or `NEGATIVE-EDGE`; (6) a list of symbols, called *state variables*; (7) a list of forms for computing values of state variables in terms of their previous values and the values of the inputs; (8) a minimum admissible clock period; (9) a list of setup times, corresponding to the inputs; and (10) a list of hold times, corresponding to the inputs. The first four of these components have the same form as the components of a combinational module, except that the variables occurring in the output forms are state variables rather than input signals. Also, a sequential module is required to have at least one input, the first of which is always interpreted as the clock input.

A simple example of a positive-edge-triggered sequential module is the following:

```
(DEFN D-FLIP-FLOP ()
  '(SEQUENTIAL          ;type
    (CLK D)             ;inputs
    (Q (M-NOT Q))       ;outputs
    (INERTIAL INERTIAL) ;modes
    ((4000 . 6000)      ;delays
     (4000 . 6000))
    POSITIVE-EDGE       ;trigger
    (Q)                 ;state variable
    (D)                 ;state form
    12000               ;period
    (6000 4000)         ;setups
    (6000 4000)))       ;holds
```

This module has two inputs: the clock input `'CLK` and a data input `'D`. It has a single state variable, `'Q`, the value of which is computed simply as the value of `'D`, and two outputs, both with inertial delay, whose values are those of `'Q` and its negation.

A setup time is given for each input. (In the above example the setups `6000` and `4000` correspond to the inputs `'CLK` and `'D`, respectively.) Each of these represents the minimum period during which the corresponding input is required to remain constant immediately *before* a triggering edge, when the value of the clock input changes from low to high (i.e., from `'F` to `'T`) for a positive-edge-triggered device, or from high to low for a negative-edge-triggered device. Thus, the first setup, corresponding to the clock input itself, is the parameter that is conventionally called the *clock low* (in the positive-edge case) or *clock high* (in the negative-edge case).

Similarly, each hold time represents the minimum period during which the corresponding input is required to remain constant immediately *after* a triggering edge; the hold time for the first input is conventionally called the *clock high* or *low*, in the positive- and negative-edge cases, respectively.

The minimum clock period is the minimum required elapsed time between successive triggering edges. In the above example, the minimum period of `12000` happens to coincide with the sum of the clock high and low times, but this need not be the case (see Subsection 5.2).

For a sequential module `mod`, the computation of `(STEP mod inp outp t)` involves the computation of the *state* of `mod` at time `t` as determined by `inp`. This state is an assignment of values to the state variables of `mod`. It is a recursive function of `t`, behaving as follows: The state of `mod` at time `'(0 . 0)` is the *null state*, which assigns the value `'X` to each state variable. As long as the inputs are well-behaved, the state changes only when a triggering edge occurs, at which time a new state is computed from the state forms, using the previous state values and the current input values. On the other hand, if at any time any input changes in violation of a setup or hold time, then the state becomes null.

Execution of `(STEP mod inp outp t)` for a sequential `mod` is the same as for a combinational module, except that the values that are posted on the output waveforms depend on both the current input values and the current state, where the latter in turn must be computed from the input history.

As an example, we shall trace the behavior of the state variable `'Q` of the D-flip-flop in response to a sample input packet. For the clock waveform, we take the following well-behaved clock pulse `w1`, which exhibits a regular period of 20000 over the interval from `'(0 . 0)` to `'(110000 . 0)`:

```
'((F . (110000 . 0)) (T . (100000 . 0))
  (F . (90000 . 0)) (T . (80000 . 0))
  (F . (70000 . 0)) (T . (60000 . 0))
  (F . (50000 . 0)) (T . (40000 . 0))
  (F . (30000 . 0)) (T . (20000 . 0))
  (F . (10000 . 0)) (T . (0 . 0)))
```

For the data input, we take the following waveform `w2`:

```
'((T . (59000 . 0)) (F . (30000 . 0)) (T . (0 . 0))).
```

Thus, the value of the input signal `'D` is `'F` on the (half-open) interval from `'(30000 . 0)` to `'(59000 . 0)` and `'T` at all other times. The value of `'Q`, which is initially `'X`, becomes `'T` at the first positive-edge (at time `'(20000 . 0)`). Since the value of `'D` changes to `'F` at time `'(30000 . 0)`, this becomes the new value of `'Q` at the next triggering edge (at `'(40000 . 0)`). The `'D` value changes again at `'(59000 . 0)`, but at the following edge (at `'(60000 . 0)`), the `'D` setup time is violated, so `'Q` becomes `'X`. This state persists until the next edge (at `'(80000 . 0)`), when the final value `'T` is assumed.

3.3 Structural Modules

A structural module has five components:

(1) a list of global input signal names;

(2) a list of submodules, which may be of any type, including the structural type;

(3) corresponding to each submodule, a list of output signal names;

(4) corresponding to each submodule, a list of input signal names, each of which is either an output of some submodule or a global input;

(5) a list of global output signal names, each of which is an output of some submodule.

Structural modules may be conveniently defined by means of the `DEFCIRCUIT` macro. For example, the following represents a D-flip-flop constructed by cross-coupling nand gates (where the module `NAND3` is a 3-input nand gate with a definition similar to that of `NAND`), as shown in Figure 1:

```
(DEFCIRCUIT D-WITH-NANDS
  (CLK D) ;inputs
  (Q QN)  ;outputs
  ((NAND) (B2 B1) (A1))
  ((NAND) (A1 CLK) (B1))
  ((NAND3) (B1 CLK B2) (A2))
  ((NAND) (A2 D) (B2))
  ((NAND) (B1 QN) (Q))
  ((NAND) (Q A2) (QN)))
```

Aside from simple syntactic requirements for the lists of input and output signals, there is only one restriction on the structure of a circuit: we allow no zero-delay cycles. (For a formal statement of this restriction, see the definition of `DELTA-ACYCLIC`.) The purpose of this restriction is to guarantee that the simulation of a structural module always terminates.

The `STEP` function is defined so that it accepts structural as well as behavioral modules as its first argument `mod`. If `mod` is structural, however, the

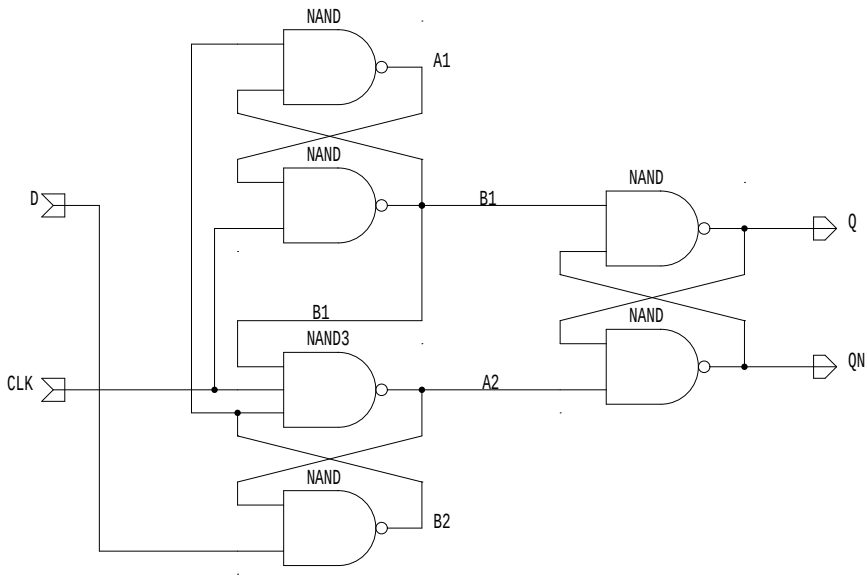


Figure 1: D-flip-flop

third argument is more complicated. In general, instead of a simple wave packet, the expected argument is an object called a *bundle* for `mod`. This notion is defined recursively as follows: if `mod` is a behavioral module, then a bundle for `mod` is just an output packet for `mod`; if `mod` is structural, then a bundle for `mod` is a list consisting of a bundle for each of its submodules. Thus, a bundle for `mod` is a list structure consisting of a waveform corresponding to each of the signals produced by `mod`. In particular, a bundle for `mod` always determines an output packet for `mod`, namely, the list of waveforms that correspond to the output signals of `mod`.

The `STEP` function is also defined recursively according to the structure of `mod`: if `inp` is an input packet and `bun` is a bundle for a structural module `mod`, then `(STEP mod inp bun t)` is the bundle for `mod` whose i^{th} member is `(STEP` $\texttt{mod}_i$ $\texttt{inp}_i$ $\texttt{bun}_i$ `t)`, where

(a) $\texttt{mod}_i$ is the ith submodule of `mod`,

(b) $\texttt{inp}_i$ is a list of the waveforms corresponding to the input signals to $\texttt{mod}_i$, extracted from `inp` and `bun` through analysis of `mod`, and

(c) $\texttt{bun}_i$ is the ith member of `bun`.

4 Simulation

4.1 The Function `SIM`

`SIM` is a function of four arguments: (1) a module `mod`, (2) a packet `inp` of waveforms corresponding to the module's inputs, (3) a time `tf` at which the simulation is to terminate, and (4) a bound `d` on the delta components of all event times. The returned value is a packet of output waveforms that is produced by simulating the module over the time interval from the origin `'(0 . 0)` to time `tf`.

In order to describe this process more precisely, let $\texttt{t}_1, \texttt{t}_2, \ldots, \texttt{t}_n$ be the increasing sequence of all times between `'(0 . 0)` $= \texttt{t}_1$ and $\texttt{tf} = \texttt{t}_n$ that have delta components not exceeding `d`. The computation of `(SIM mod inp tf d)` involves a call to `STEP` corresponding to each of these times: Let $\texttt{bun}_0$, the initial bundle for the simulation, be the bundle for `mod` in which every waveform has the constant value `'X` (i.e, every waveform is the alist

'((X . (0 . 0)))). For $i = 1, \ldots, n$, let $\texttt{bun}_i$ be the value of (STEP mod inp $\texttt{bun}_{i-1}$ $\texttt{t}_i$). The value of (SIM mod inp tf d) is the output packet determined by the bundle $\texttt{bun}_n$.

4.2 Delta Constraints

Note that the delta bound d is required to reduce the set of times within a given interval to a finite set, and thus to guarantee termination of the recursive function SIM. In order to produce the intended behavior of this function, we must impose constraints on its arguments that ensure that the times of all scheduled events have delta components bounded by d.

This will require several definitions. First, we shall say that a waveform w is *bounded* by d if no event time occurring in w has delta component exceeding d. Next, we define the *level* of a signal in a circuit to be the maximum of the lengths of all zero-delay paths through the circuit starting at the given signal. A waveform w is *admissible* for a signal s with respect to d if $\ell \leq \texttt{d}$ and w is bounded by $\texttt{d} - \ell$, where ℓ is the level of s. A bundle, or similarly, an input packet, for mod is *admissible* with respect to d if each of its waveforms is admissible for the signal with which it is associated.

Finally, we may state the following important result: If inp and bun are an admissible input packet and an admissible bundle for mod wrt d, respectively, then (STEP mod inp bun t) is an admissible bundle for mod wrt d. It follows that if inp is an admissible input packet for mod wrt d, then for any time t, every waveform in the bundle (SIM mod inp t d) is bounded by d.

It should be noted that our primary motivation for including delta delays in our language, in spite of the inherent complications described above, is a commitment to adhere to the VHDL delay model. All of the modules that we have defined in this language, including all of the examples presented herein, exhibit only positive delays. Thus, for our purposes, we may always take the d parameter of SIM to be 0, and need never deal with times with nonzero delta components.

4.3 Efficient Execution

The definition of the function SIM, as described at the beginning of this section, is designed to be as theoretically simple as possible. Its execution, on the other hand, is impractical for two reasons:

1. Every call to STEP for a sequential module requires complete calculation of the module's state from its input packet.

2. STEP is called at every legal time during the simulation interval, although it has no effect at times when no events are scheduled.

For the purpose of execution, therefore, we have defined a more efficient function, FAST-SIM, which may be shown to be equivalent to SIM. This efficiency is achieved by eliminating both aspects of the redundancy noted above. Firstly, it is defined in terms of a function FAST-STEP, which records the states of sequential modules, so that at each step, a state need only be updated rather than entirely recomputed. Secondly, FAST-SIM is truly event-driven: it calls FAST-STEP only at times when a relevant event is scheduled.

As an illustration, let us consider a call to SIM with the following arguments: (1)the sequential module D-FLIP-FLOP; (2)the input packet consisting of the waveforms w1 and w2, defined in Subsection 3.2; (3) the terminal time '(200000 . 0); (4) the delta bound 0. This results in 200001 calls to STEP (this number would be even larger if we changed the fourth argument). Each of these calls requires a recomputation of the state by reexamining the entire input history, which is clearly impractical. The execution of FAST-SIM on the same arguments, on the other hand, involves only 18 calls to FAST-STEP, each of which requires only updating the state in response to the most recent events. The value returned by

```
(FAST-SIM (D-FLIP-FLOP) (LIST w1 w2) '(200000 . 0) 0)
```

is the output packet

```
'(((T 86000 . 0) (X 64000 . 0) (F 46000 . 0)
   (X 44000 . 0) (T 26000 . 0) (X 0 . 0))
  ((F 86000 . 0) (X 64000 . 0) (T 46000 . 0)
   (X 44000 . 0) (F 26000 . 0) (X 0 . 0))).
```

Note that these waveforms record the behavior of the two output signals whose (delayed) values are defined to be those of the state variable 'Q and its negation. It is instructive to compare this result with the trace of 'Q given in Subsection 3.2.

5 Specification

Let `modc` and `moda` be two modules. We shall say that `modc` is an *implementation* of `moda`, or equivalently, that `moda` is a *specification* of `modc`, if the following relation holds: Given a number `d`, a time `t`, and an input packet `inp` for `moda` wrt `d`, `inp` is also an input packet for `modc` wrt `d` and `(SIM moda inp t d)` generalizes `(SIM modc inp t d)`. If one module is both an implementation and a specification of the other, then we say that the two are *equivalent*.

This notion of specification is central to our approach to circuit verification. Our goal is to characterize the behavior of circuits by deriving behavioral modules that are specifications of given structural modules. For example, the correctness of our flip-flip implementation `D-WITH-NANDS` can be established by showing that it is an implementation in the above sense of the sequential module `D-FLIP-FLOP`. The proof of this theorem remains to be mechanically checked, but we may illustrate it by comparing simulations of the two modules on the same input. Thus, when `(D-FLIP-FLOP)` is replaced with `(D-WITH-NANDS)` as the first argument in the call to `FAST-SIM` appearing in the previous section, the following output packet is returned:

```
'(((T 67000 . 0) (F 46000 . 0) (T 24000 . 0) (X 0 . 0))
  ((F 69000 . 0) (T 44000 . 0) (F 26000 . 0) (X 0 . 0)))
```

It is easily seen that this output of the implementation `D-WITH-NANDS` is indeed generalized by that of the specification `D-FLIP-FLOP` . It is also worth noting that this simulation of the implementation involves 35 calls to `FAST-STEP` on `D-WITH-NANDS` along with 76 calls to `STEP` on its combinational submodules, as compared to only 18 calls to `FAST-STEP` for `D-FLIP-FLOP`. Thus, there are two distinct benefits of establishing specifications for modules: concise behavioral description and efficient simulation.

In order to facilitate the verification of more complex circuits, we shall require the following basic results:

1. If `formc` is one of the output forms of a behavioral module `modc`, `forma` is another form such that the value of `forma` generalizes the value of `formc` for any assignment of variable values, and `moda` is the result of replacing `formc` in `modc` with `forma`, then `modc` implements `moda`.

2. If `modc` implements `moda`, and `structc` is the result of replacing an occurrence of `moda` in `structa` with `modc`, then `structc` implements `structa`.

3. If `mod1` is a structural module that contains a structural module `sub` as a submodule, and `mod2` is the result of "flattening" `mod1` by replacing the occurrence of `sub` with the list of submodules of `sub` (and reconstructing all input and output lists accordingly), then `mod1` and `mod2` are equivalent.

4. If `modc` is a behavioral module that has an output with either `TRANSPORT` or `INERTIAL` delay mode, and `moda` is the result of changing this delay mode to `NONDETERMINISTIC`, then `modc` implements `moda`.

5. If `modc` is a behavioral module that has an output with delay range `(min1 . max1)`, and `moda` is the result of replacing that delay range with `(min2 . max2)`, where `min2` $\leq$ `min1` and `max2` $\geq$ `max1`, then `modc` implements `moda`.

The first of these results is trivial, and its application often amounts to mere tautology checking: if a complicated output form may be shown to be logically equivalent to a simpler form, then the simpler form may be substituted without affecting functionality.

The second result is significant in that it provides for hierarchical circuit analysis: Suppose we wish to analyze a complex structural module that has structural components. Once behavioral specifications are derived for the components, they may be substituted to yield a specification for the original structure, which is a step toward its behavioral specification.

As applications of the other three results, we have implemented two procedures for deriving behavioral specifications (combinational and sequential, respectively) for certain classes of structural modules. These are described below.

5.1 Combinational Specifications

Any structural module `modc` that (a) is constructed entirely of combinational components, and (b) contains no loops, may be shown to be an implementation of some behavioral combinational module `moda`. The function

COMB-REDUCE, after verifying that a given structure modc satisfies these requirements, automatically generates the appropriate specification moda, constructing its components as follows:

1. The input signals of moda are the global inputs of modc.
2. The form for each output is computed by tracing backwards from each output, constructing by means of a series of substitutions an expression for the output value in terms of inputs alone.
3. The delay range for each output is determined by the minimum and maximum of the total delays along all paths from inputs.
4. The delay mode for every output is NONDETERMINISTIC.

The following 1-bit adder, built out of nand gates as shown in Figure 2, is an example of a circuit that meets the above requirements:

```
(DEFCIRCUIT ADDER1
  (A B C-IN)                    ;inputs
  (S C-OUT)                     ;outputs
  ((NAND) (A B) (T1))         ;i1
  ((NAND) (A T1) (T2))        ;i2
  ((NAND) (B T1) (T3))        ;i3
  ((NAND) (T2 T3) (T4))       ;i4
  ((NAND) (C-IN T4) (T5))     ;i5
  ((NAND) (C-IN T5) (T7))     ;i6
  ((NAND) (T5 T4) (T6))       ;i7
  ((NAND) (T5 T1) (C-OUT))    ;i8
  ((NAND) (T7 T6) (S)))       ;i9
```

The intended behavior of this device may be described in terms of the functions M-SUM3 and M-MAJ3, which compute the sum modulo 2 and the majority, respectively, of three bits. Assuming that the inputs A, B, and C-IN remain stable for a sufficiently long period, the outputs S and C-OUT of ADDER1 should eventually stabilize with the values (M-SUM3 A B C-IN) and (M-MAJ3 A B C-IN), respectively.

As a first step toward a verified formalization of this description, we apply COMB-REDUCE to ADDER1, computing the following specification:

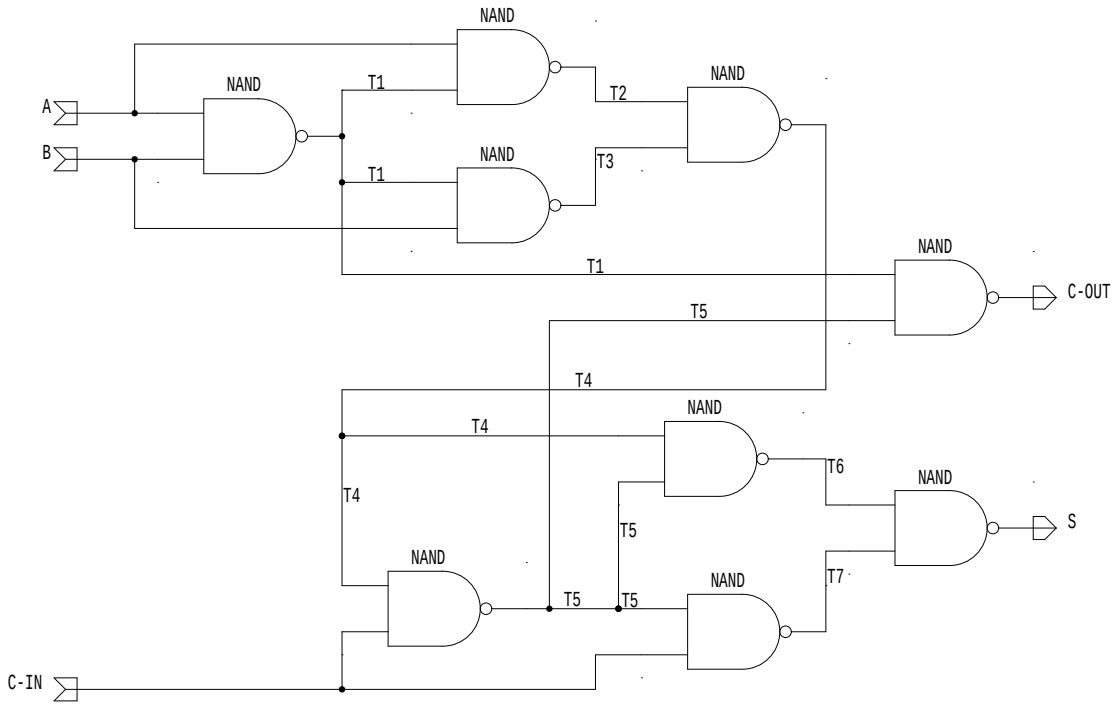


Figure 2: 1-Bit adder

```
'(COMBINATIONAL
  (A B C-IN)
  ((M-NAND (M-NAND C-IN
                   (M-NAND C-IN
                           (M-NAND (M-NAND A (M-NAND A B))
                                   (M-NAND B (M-NAND A B)))))
           (M-NAND (M-NAND C-IN
                          (M-NAND (M-NAND A (M-NAND A B))
                                  (M-NAND B (M-NAND A B))))
                   (M-NAND (M-NAND A (M-NAND A B))
                           (M-NAND B (M-NAND A B)))))
   (M-NAND (M-NAND C-IN
                   (M-NAND (M-NAND A (M-NAND A B))
                           (M-NAND B (M-NAND A B))))
           (M-NAND A B)))
  (NONDETERMINISTIC NONDETERMINISTIC)
  ((4000 . 12000) (4000 . 10000)))
```

The two outputs of this module will stabilize after maximum delays of `12000` and `10000`, respectively, assuming stable inputs. Their values, however, are given by rather complicated expressions in terms of `M-NAND`. To complete our analysis of `ADDER1`, we must show that these two expressions are tautologically equivalent to the forms `(M-SUM A B C-IN)` and `(M-MAJ3 A B C-IN)`. Once this is done (automatically by the Nqthm prover), we may conclude that the following is a specification for `ADDER1`:

```
'(COMBINATIONAL
  (A B C-IN)
  ((M-SUM3 A B C-IN)
   (M-MAJ3 A B C-IN))
  (NONDETERMINISTIC NONDETERMINISTIC)
  ((4000 . 12000) (4000 . 10000)))
```

5.2 Sequential Specifications

Our algorithm for deriving a sequential behavioral specification of a structural module with sequential components requires that (a) the structure contains no cycles passing only through combinational components, (b) all global

outputs are expressible as functions of state alone (and not of global inputs), (c) all sequential submodules have the same trigger and are connected to the same clock input, and (d) the minimum delays of the outputs of the sequential components are long enough to respect the hold times of any sequential inputs to which they are connected (either directly or through paths consisting only of combinational components). The function `SEQ-REDUCE`, after verifying that a given structure `modc` satisfies these requirements, automatically generates a behavioral specification `moda`. As a preliminary step, the signals of `modc` and its submodules are all renamed in order to avoid any conflicts. The components of `moda` are then derived as follows:

1. The input signals of `moda` are the global inputs of `modc`.

2. The trigger of `moda` is the trigger of the sequential submodules of `modc`.

3. The state variables of `moda` are the state variables of all the sequential components of `modc`.

4. The state forms of `moda` are computed by tracing backwards from each state variable of `modc` to sequential outputs and global inputs, and constructing by means of a series of substitutions an expression for the state variable in terms of state variables and global inputs.

5. The output forms of `moda` correspond to the global outputs of `modc`; they are computed by tracing backwards from each global output to sequential outputs and constructing by means of a series of substitutions an expression for the output value in terms of state variables alone.

6. The delay range for each output is determined by the minimum and maximum of the total delays along all paths from sequential outputs.

7. The delay mode of an output is `NONDETERMINISTIC` unless it is generated directly by a sequential component of `modc`, in which case it inherits its mode from that component.

8. The setup and hold times of each input are computed as the minimum times required to respect the setup and hold times of the inputs of the sequential components to which they are connected.

9. The clock period is the maximum of the longest clock period of sequential component and the time required for internal signals to stabilize in order to respect setup times of sequential inputs.

As an extremely simple example, we consider the following module, constructed by connecting two D-flip-flops (as illustrated in Figure 3):

```
(DEFCIRCUIT DOUBLE-FLIP-FLOP
  (CLK D)                          ;inputs
  (OUT OUTN)                       ;outputs
  ((D-FLIP-FLOP) (CLK D) (Q QN))
  ((D-FLIP-FLOP) (CLK Q) (OUT OUTN)))
```

SEQ-REDUCE derives the following behavioral specification for this structure:

```
'(SEQUENTIAL                ;type
  (CLK-2 D-2)               ;inputs
  (Q-1 (M-NOT Q-1))         ;outputs
  (INERTIAL INERTIAL)       ;modes
  ((4000 . 6000)            ;delays
   (4000 . 6000))
  POSITIVE-EDGE             ;trigger
  (Q-0 Q-1)                 ;state variable
  (D-2 Q-0)                 ;state form
  12000                     ;period
  (6000 4000)               ;setups
  (6000 4000)))             ;holds
```

Note, however, that the structure only barely satisfies the last item in our list of preconditions for SEQ-REDUCE, since the minimum output delay of D-FLIP-FLOP happens to coincide with the setup time of 4000. That is, if the definition of the flip-flop were altered by replacing the lower limit of the first delay range by any number smaller than 4000, then DOUBLE-FLIP-FLOP would be rejected by SEQ-REDUCE.

Our final example is a 4-bit loadable shift register composed of nand gates and D-flip-flops. We define this structure hierarchically, as shown in Figure 4, using a component consisting of three gates and a flip-flop:

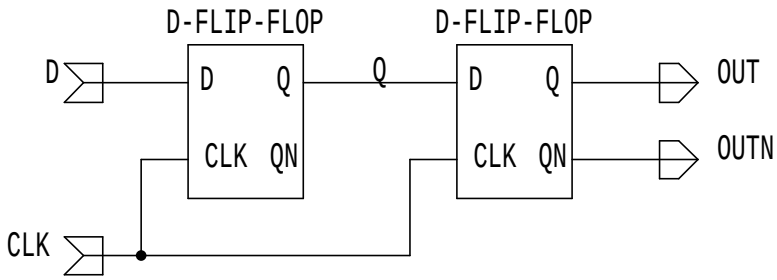


Figure 3: DOUBLE-FLIP-FLOP

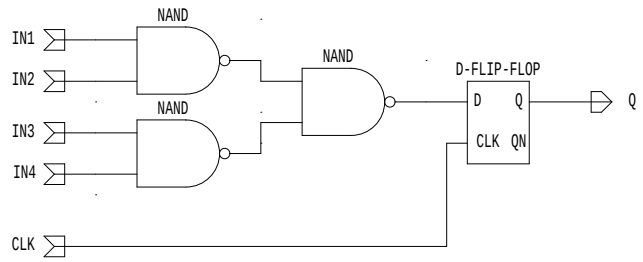
```
(DEFCIRCUIT SHIFTER-COMPONENT
  (CLK IN1 IN2 IN3 IN4)        ;inputs
  (Q)                          ;outputs
  ((NAND) (IN1 IN2) (S1))
  ((NAND) (IN3 IN4) (S2))
  ((NAND) (S1 S2) (D))
  ((D-FLIP-FLOP) (CLK D) (Q QBAR)))
```

The register is constructed from four of these components:

```
(DEFCIRCUIT SHIFTER
  (CLK LOAD AIN BIN CIN DIN)      ;inputs
  (AOUT BOUT COUT DOUT)           ;outputs
  ((INV) (LOAD) (SHIFT))
  ((SHIFTER-COMPONENT) (CLK DOUT SHIFT AIN LOAD) (AOUT))
  ((SHIFTER-COMPONENT) (CLK AOUT SHIFT BIN LOAD) (BOUT))
  ((SHIFTER-COMPONENT) (CLK BOUT SHIFT CIN LOAD) (COUT))
  ((SHIFTER-COMPONENT) (CLK COUT SHIFT DIN LOAD) (DOUT)))
```

The following behavioral specification is generated by `SEQ-REDUCE`:

```
'(SEQUENTIAL
  (CLK-5 LOAD-5 AIN-5 BIN-5 CIN-5 DIN-5)  ;inputs
  (Q-3-1 Q-3-2 Q-3-3 Q-3-4)               ;outputs
  (INERTIAL INERTIAL INERTIAL INERTIAL)   ;modes
  ((4000 . 6000) (4000 . 6000)            ;delays
   (4000 . 6000) (4000 . 6000))
  POSITIVE-EDGE                           ;trigger
  (Q-3-1 Q-3-2 Q-3-3 Q-3-4)               ;state variables
  ((M-NAND (M-NAND Q-3-4 (M-NOT LOAD-5))  ;state forms
           (M-NAND AIN-5 LOAD-5))
   (M-NAND (M-NAND Q-3-1 (M-NOT LOAD-5))
           (M-NAND BIN-5 LOAD-5))
   (M-NAND (M-NAND Q-3-2 (M-NOT LOAD-5))
           (M-NAND CIN-5 LOAD-5))
   (M-NAND (M-NAND Q-3-3 (M-NOT LOAD-5))
           (M-NAND DIN-5 LOAD-5)))
```



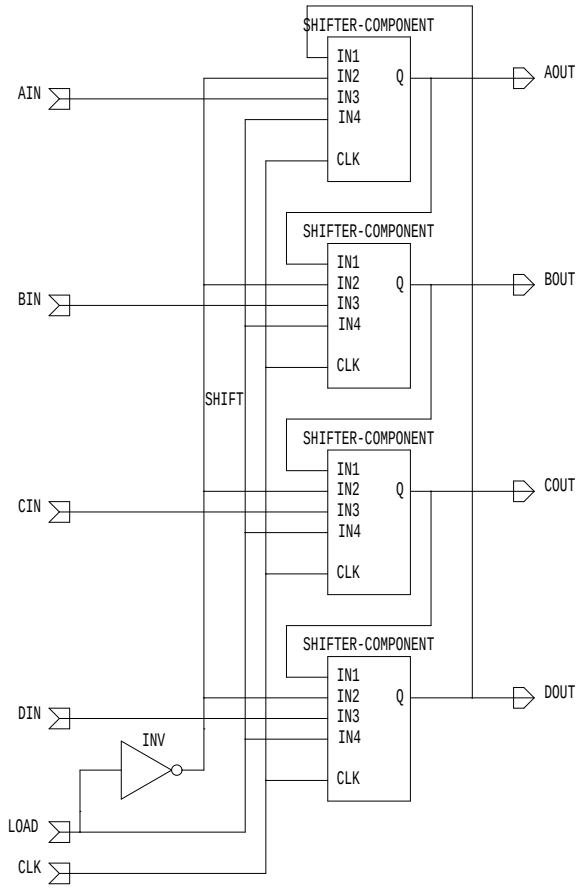


Figure 4: Loadable shift register

```
14000                                   ;period
(6000 10000 8000 8000 8000 8000)        ;setups
(6000 0 0 0 0 0))                       ;holds
```

This sequential module has four state variables and four matching outputs, corresponding to the four flip-flops. It also has four "data" inputs, along with a clock and a "load" input. On each cycle, a new state is computed as follows: if the load is high, then each state variable assumes the value of the corresponding input; if the load is low, then the values of the state variables are rotated. Although this behavior may be difficult to ascertain from the state forms shown above, it becomes clear once the following tautology is noted:

```
(EQUAL (M-NAND (M-NAND Q (M-NOT LOAD))
               (M-NAND A LOAD))
       (M-OR (M-AND LOAD A)
             (M-AND (M-NOT LOAD) Q))).
```

This is our only example of a sequential module with a minimum clock period (`14000`) that exceeds the sum of the clock setup and hold times (`12000`). The reason for this is that a signal that is sent from one flip-flop to another must arrive sufficiently in advance of a triggering edge to respect the receiver's setup time. Thus, the time elapsed from one positive edge to the next must be at least the sum of the maximum delay of the sent signal (`6000`), the delay along the path to the receiver (`4000`), and the setup time of the receiver's input (`4000`).

It is also worth noting that the hold times for all but the clock input are `0`. The reason for this is that the delay along every path from an input to a flip-flop is at least as long as the flip-flop's hold time.

6 Future Work

The HDL that we have described is sufficiently expressive for the modeling of both synchronous and asynchronous devices. Thus far, however, we have only outlined a methodology for specifying and verifying combinational and synchronous circuits designed in this language. Many of the theorems on which this methodology is based remain to be formalized and mechanically

checked. Once this body of theorems is established, our next goal will be to extend the theory to the asynchronous realm. This effort will be driven by the design of a circuit that achieves communication between two asynchronous processors according to a version of the protocol that was formalized in [7]. The formal specification and verification of this design will be delivered with the report on Task 5.

References

[1] Bevier, William R. and Young, William D., Machine checked proofs of the Design and Implementation of a Fault-Tolerant Circuit, Technical Report 62, Computational Logic, Inc., NASA CR-182099, November 1990.

[2] Borrione, Dominique D., Pierre, Laurence V., and Salem, Ashraf M., Formal verification of VHDL descriptions in the PREVAIL environment, in *IEEE Design and Test*, June, 1992.

[3] Boyer, R. S. and Moore, J, *A Computational Logic Handbook*, Academic Press, Boston, 1988.

[4] Brock, Bishop C. and Hunt, Warren A., Jr., A Formal HDL and its use in the FM9001 verification, in *Proceedings of the Royal Society*, 1992.

[5] Brock, Bishop C., Hunt, Warren A., Jr., and Young, William D., Introduction to a formally defined hardware description language. In *Proceedings of the IFIP Conference on Theorem Provers in Circuit Design*, June 1992.

[6] Filippenko, Ivan V., VHDL verification in the State Delta Verification System, in *ACM SIGDA International Workshop on Formal Methods in VLSI Design*, January 1991.

[7] Institute of Electrical and Electronic Engineers, *IEEE Standard VHDL Language Reference Manual*, 1988.

[8] Moore, J Strother, A Formal model of asynchronous communication and its use in mechanically verifying a biphase mark protocol, Technical Report 68, Computational Logic, Inc., NASA CR-4433, June 1992.

[9] Young, William D., Verifying the interactive convergence clock synchronization algorithm using the Boyer-Moore theorem prover, Technical Report 77, Computational Logic, Inc., NASA CR-189649, April 1992.

Appendix

```
;*********************************************************************
;;                          WAVEFORMS
;;*********************************************************************

;;A moment in time is a pair of numbers:

(defn timep (x)
  (and (listp x)
       (numberp (car x))
       (numberp (cdr x))))

(defn zero-time ()
  '(0 . 0))

;;Moments in time are ordered lexicographically:

(defn tlessp (a b)
  (if (equal (car a) (car b))
      (lessp (cdr a) (cdr b))
    (lessp (car a) (car b))))

(defn tleq (a b)
  (not (tlessp b a)))

;;Events are scheduled at times that are computed from the current time
;;and propagation delays as follows:

(defn tplus (t0 delay)
  (if (zerop delay)
      (cons (car t0) (add1 (cdr t0)))
    (cons (plus (car t0) delay) 0)))


;;A waveform is an alist that associates signal values with the times
;;at which they are assumed by the signal:
```

```
(defn waveformp (w)
  (if (listp w)
      (if (listp (cdr w))
          (and (waveformp (cdr w))
               (timep (cdar w))
               (tlessp (cdadr w) (cdar w))
               (not (equal (caadr w) (caar w))))
        (equal (cdar w) (zero-time)))
    f))

;;A packet is a list of waveforms:

(defn packetp (l n)
  (if (zerop n)
      (nlistp l)
    (and (listp l)
         (waveformp (car l))
         (packetp (cdr l) (sub1 n)))))

;;The value of a signal at a given time is computed from its waveform
;;as follows:

(defn wave-value (wave time)
  (if (listp wave)
      (if (tlessp time (cdar wave))
          (wave-value (cdr wave) time)
        (caar wave))
    f))

(defn packet-values (packet time)
  (if (listp packet)
      (cons (wave-value (car packet) time)
            (packet-values (cdr packet) time))
    ()))

;;To compute the final value of a waveform:

(defn last-value (w)
  (caar w))

(defn last-values (p)
  (if (listp p)
      (cons (last-value (car p))
```

```
            (last-values (cdr p)))
    ()))

;;There is no restriction on the values that may be assumed by a signal.
;;The value 'X, however, is special -- it represents an unknown value.
;;Any other value is "definite".  A waveform is definite it never assumes
;;the value 'X:

(defn defvalp (v)
  (not (equal v 'x)))

(defn defwavep (w)
  (if (listp w)
      (and (defvalp (caar w))
           (defwavep (cdr w)))
    t))

;;A value v1 generalizes v2 if v1 is either v2 or 'X.  A wave w1
;;generalizes a wave w2 if at all times, the value of w1 generalizes the
;;value of w2:

(defn genvalp (v1 v2)
  (or (equal v1 v2) (equal v1 'x)))

(defn genwavep (w1 w2)
  (if (and (listp w1) (listp w2))
      (and (genvalp (caar w1) (caar w2))
           (if (tlessp (cdar w2) (cdar w1))
               (genwavep (cdr w1) w2)
             (if (tlessp (cdar w1) (cdar w2))
                 (genwavep w1 (cdr w2))
               (or (equal (cdar w1) (zero-time))
                   (genwavep (cdr w1) (cdr w2))))))
    f)
  ((lessp (plus (count w1) (count w2)))))

(defn genpacketp (p1 p2)
  (if (listp p1)
      (and (genwavep (car p1) (car p2))
           (genpacketp (cdr p1) (cdr p2)))
    (nlistp p2)))

;;Histories and futures:
```

```
(defn wave-history (wave time)
  (if (listp wave)
      (if (tlessp time (cdar wave))
          (wave-history (cdr wave) time)
        wave)
    wave))

(defn packet-history (packet time)
  (if (listp packet)
      (cons (wave-history (car packet) time)
            (packet-history (cdr packet) time))
    ()))

(defn packet-histories (packets t0)
  (if (listp packets)
      (cons (packet-history (car packets) t0)
            (packet-histories (cdr packets) t0))
    ()))

(defn wave-future (wave time)
  (if (listp wave)
      (if (tlessp time (cdar wave))
          (cons (car wave) (wave-future (cdr wave) time))
        (if (tlessp (cdar wave) time)
            (list (cons (caar wave) time))
          (list (car wave))))
    wave))

(defn packet-future (packet time)
  (if (listp packet)
      (cons (wave-future (car packet) time)
            (packet-future (cdr packet) time))
    ()))

;;To determine whether some waveform of a packet aquires a new value
;;at a given time:

(defn new-value-p (wave time)
  (if (listp wave)
      (if (tlessp time (cdar wave))
          (new-value-p (cdr wave) time)
        (equal time (cdar wave)))
    f))
```

```
(defn some-new-value-p (packet time)
  (if (listp packet)
      (or (new-value-p (car packet) time)
          (some-new-value-p (cdr packet) time))
    f))


;;*********************************************************************
;;                          PROPAGATION
;;*********************************************************************

;;The following two functions implement "transport" and "inertial"
;;delay, as defined in the VHDL standard.  They may be used to schedule
;;a transaction with value V at time T1 on a waveform W, assuming that V
;;and all values of W are definite, and that T1 exceeds the current time
;;T0:

(defn post-transport-event-definite (w v t1)
  (if (listp w)
      (if (tlessp (cdar w) t1)
          (if (equal (caar w) v)
              w
            (cons (cons v t1) w))
        (post-transport-event-definite (cdr w) v t1))
    f))

(defn post-inertial-event-definite (w v t0 t1)
  (if (listp w)
      (if (tlessp t0 (cdar w))
          (if (and (tlessp (cdar w) t1) (equal v (caar w)))
              (post-inertial-event-definite (cdr w) v t0 (cdar w))
            (post-inertial-event-definite (cdr w) v t0 t1))
        (if (equal v (caar w))
            w
          (cons (cons v t1) w)))
    f))

;;In the presence of indefinite values, we use the following more
;;general functions.  Instead of fixed delays, we allow delay ranges:
;;we assume that the time of the event is at most T2 and (if T1
;;precedes T2) at least T1, where T12 and T2 both exceed T0:

(defn post-transport-event (w v t0 t1 t2)
  (if (listp w)
```

```
  (if (tlessp t0 (cdar w))
      (if (tlessp (cdar w) t2)
          (if (equal v (caar w))
              (post-transport-event (cdr w) v t0 t1 (cdar w))
            (if (tlessp t1 t2)
                (if (tlessp t1 (cdar w))
                    (if (listp (cdr w))
                        (if (equal v (caadr w))
                            (if (equal v 'x)
                                (cdr w)
                              (cons (cons v t2)
                                    (cons (cons 'x (cdar w))
                                          (post-transport-event
                                            (cddr w) v t0
                                            t1 (cdadr w)))))
                          (post-transport-event (cdr w) v t0 t1 t2))
                      f)
                  (if (tlessp (cdar w) t1)
                      (if (equal (caar w) 'x)
                          (cons (cons v t2) w)
                        (if (equal v 'x)
                            (cons (cons 'x t1) w)
                          (cons (cons v t2) (cons (cons 'x t1) w))))
                    (if (listp (cdr w))
                        (if (equal (caadr w) 'x)
                            (cons (cons v t2) (cdr w))
                          (if (equal v 'x)
                              (cons (cons 'x t1) w)
                            (cons (cons v t2)
                                  (cons (cons 'x t1) (cdr w)))))
                      f)))
              (cons (cons v t2) w)))
        (post-transport-event (cdr w) v t0 t1 t2))
    (if (equal (caar w) v)
        w
      (if (equal (caar w) 'x)
          (cons (cons v t2) w)
        (if (tlessp t1 t2)
            (if (equal v 'x)
                (cons (cons 'x t1) w)
              (cons (cons v t2) (cons (cons 'x t1) w)))
          (cons (cons v t2) w)))))
f))
```

```
(defn post-inertial-event (w v t0 t1 t2)
  (if (listp w)
      (if (tlessp t0 (cdar w))
          (if (tlessp (cdar w) t2)
              (if (equal v (caar w))
                  (post-inertial-event (cdr w) v t0 t1 (cdar w))
                (if (and (tlessp (cdar w) t1)
                         (or (equal (caar w) 'x)
                             (and (equal v 'x)
                                  (not (equal (caar w) (cdadr w))))))
                    (post-inertial-event (cdr w) v t0 (cdar w) t2)
                  (post-inertial-event (cdr w) v t0 t1 t2)))
            (post-inertial-event (cdr w) v t0 t1 t2))
        (if (equal (caar w) v)
            w
          (if (equal (caar w) 'x)
              (cons (cons v t2) w)
            (if (tlessp t1 t2)
                (if (equal v 'x)
                    (cons (cons 'x t1) w)
                  (cons (cons v t2) (cons (cons 'x t1) w)))
              (cons (cons v t2) w)))))
    f))

;;We also provide a third delay mode, NONDETERMINISTIC, which generalizes
;;both TRANSPORT and INERTIAL:

(defn post-nondeterministic-event (w v t0 t1 t2)
  (if (listp w)
      (if (tlessp t0 (cdar w))
          (if (tlessp (cdar w) t1)
              (if (listp (cdr w))
                  (if (equal (caar w) (caadr w))
                      (post-nondeterministic-event (cdr w) v t0 t1 t2)
                    (post-nondeterministic-event
                      (cdr w) v t0 (cdar w) t2))
                f)
            (post-nondeterministic-event (cdr w) v t0 t1 t2))
        (if (or (equal (caar w) 'x) (tleq t2 t1))
            (if (equal (caar w) v)
                w
              (cons (cons v t2) w))
          (if (equal v 'x)
              (cons (cons 'x t1) w)
```

```
            (cons (cons v t2) (cons (cons 'x t1) w)))))
    f))


;;*********************************************************************
;;                              MODULES
;;*********************************************************************

(defn type (mod)
  ;a litatom
  (car mod))

(disable type)

;We shall implement three module types, COMBINATIONAL, SEQUENTIAL, and
;;STRUCTURAL. Combinational and sequential modules are called BEHAVIORAL.

(defn combinationalp (mod)
  (equal (type mod) 'combinational))

(defn sequentialp (mod)
  (equal (type mod) 'sequential))

(defn behavioralp (mod)
  (or (combinationalp mod) (sequentialp mod)))

(defn structuralp (mod)
  (equal (type mod) 'structural))

;;Associated with any module are lists of inputs and outputs:

(defn inputs (mod)
  (cadr mod))

(disable inputs)

(defn outputs (mod)
  (caddr mod))

(disable outputs)

(defn number-of-inputs (mod)
  (length (inputs mod)))
```

```
(defn number-of-outputs (mod)
  (length (outputs mod)))

;;Module behavior will be characterized by a "step" function of 4
;;arguments: (1) a module, (2) an input packet, (3) an output packet,
;;and (4) a time.  The value returned is the result of updating the
;;output packet by executing any events in the input packet that occur
;;at the given time.  This function will be required to exhibit the
;;following five properties (although I don't know why I care about
;;the last two):


;;  (1) Monotonic:

;;           (IMPLIES (AND (PACKETP INP1 (NUMBER-OF-INPUTS mod))
;;                         (PACKETP INP2 (NUMBER-OF-INPUTS mod))
;;                         (GENPACKETP INP1 INP2)
;;                         (PACKETP OUT1 (NUMBER-OF-OUTPUTS mod))
;;                         (PACKETP OUT2 (NUMBER-OF-OUTPUTS mod))
;;                         (GENPACKETP OUTP1 OUTP2)
;;                         (TIMEP T0))
;;                    (GENPACKETP (STEP mod INP1 OUTP1 T0)
;;                                (STEP mod INP2 OUTP2 T0)))

;;  (2) Nonpredictive:

;;           (IMPLIES (AND (PACKETP INP1 (NUMBER-OF-INPUTS mod))
;;                         (PACKETP INP2 (NUMBER-OF-INPUTS mod))
;;                         (EQUAL (PACKET-HISTORY INP1 T0)
;;                                (PACKET-HISTORY INP2 T0))
;;                         (PACKETP OUT (NUMBER-OF-OUTPUTS mod))
;;                         (TIMEP T0))
;;                    (EQUAL (STEP mod INP1 OUTP T0)
;;                           (STEP mod INP2 OUTP T0)))

;;For combinational modules, Property (2) may be strengthened as follows:

;;           (IMPLIES (AND (PACKETP INP1 (NUMBER-OF-INPUTS mod))
;;                         (PACKETP INP2 (NUMBER-OF-INPUTS mod))
;;                         (EQUAL (PACKET-VALUES INP1 T0)
;;                                (PACKET-VALUES INP2 T0))
;;                         (PACKETP OUT (NUMBER-OF-OUTPUTS mod))
;;                         (TIMEP T0))
;;                    (EQUAL (STEP mod INP1 OUTP T0)
```

```
;;                          (STEP mod INP2 OUTP T0)))

;;  (3) Nonretroactive:

;;           (IMPLIES (AND (PACKETP INP (NUMBER-OF-INPUTS mod))
;;                         (PACKETP OUT (NUMBER-OF-OUTPUTS mod))
;;                         (TIMEP T0))
;;            (EQUAL (PACKET-HISTORY (STEP mod INP OUTP T0) T0)
;;                   (PACKET-HISTORY OUTP T0))

;;  (4) Nonretrospective:

;;           (IMPLIES (AND (PACKETP INP (NUMBER-OF-INPUTS mod))
;;                         (PACKETP OUT1 (NUMBER-OF-OUTPUTS mod))
;;                         (PACKETP OUT2 (NUMBER-OF-OUTPUTS mod))
;;                         (EQUAL (PACKET-FUTURE OUTP1 T0)
;;                                (PACKET-FUTURE OUTP2 T0))
;;                         (TIMEP T0))
;;                    (EQUAL (PACKET-FUTURE (STEP mod INP OUTP1 T0) T0)
;;                           (PACKET-FUTURE (STEP mod INP OUTP2 T0) T0)))

;;  (5) Idempotent:

;;           (IMPLIES (AND (PACKETP INP (NUMBER-OF-INPUTS mod))
;;                         (PACKETP OUT (NUMBER-OF-OUTPUTS mod))
;;                         (TIMEP T0))
;;            (EQUAL (STEP mod INP (STEP mod INP OUTP T0) T0)
;;                   (STEP mod INP OUTP T0))

;;*********************************************************************
;;                   COMBINATIONAL MODULES
;;*********************************************************************

;;Associated with each output of a behavioral module is a delay mode,
;;which may be INERTIAL, TRANSPORT, or NONDETERMINISTIC, and a delay
;;range:

(defn modes (mod)
  ;a list of litatoms
  (cadddr mod))

(disable modes)

(defn delays (mod)
```

```
  ;a list of pairs of numbers, (MIN . MAX), corresponding to outputs.
  ;If MAX is NIL (more generally, if MAX does not exceed MIN), then MIN
  ;is used for both extremes.
  (caddddr mod))

(disable delays)

(defn min-delay (pair) (car pair))

(defn max-delay (pair)
  (max (car pair) (cdr pair)))

(defn post-event (w v t0 mode t1 t2)
  (case mode
    (transport (post-transport-event w v t0 t1 t2))
    (inertial (post-inertial-event w v t0 t1 t2))
    (nondeterministic (post-nondeterministic-event w v t0 t1 t2))
    (otherwise (post-inertial-event w v t0 t1 t2))))

(defn post-events (packet values t0 modes delays)
  (if (listp packet)
      (cons (post-event (car packet)
                        (car values)
                        t0
                        (car modes)
                        (tplus t0 (min-delay (car delays)))
                        (tplus t0 (max-delay (car delays))))
            (post-events (cdr packet)
                         (cdr values)
                         t0
                         (cdr modes)
                         (cdr delays)))
    ()))

(defn combinational-step (mod inp outp time)
  (post-events outp
               (eval$ 'list
                      (outputs mod)
                      (pairlist (inputs mod) (packet-values inp time)))
               time
               (modes mod)
               (delays mod)))

;;Some gates:
```

```
(defn m-and (a b)
  (if (equal a 'f) 'f
    (if (equal b 'f) 'f
      (if (and (equal a 't) (equal b 't)) 't
        'x))))

(defn m-or (a b)
  (if (equal a 't) 't
    (if (equal b 't) 't
      (if (and (equal a 'f) (equal b 'f)) 'f
        'x))))

(defn m-not (a)
  (if (equal a 't) 'f
    (if (equal a 'f) 't
      'x)))

(defn m-nand (a b)
  (m-not (m-and a b)))

(defn m-and3 (a b c)
  (if (equal a 'f) 'f
    (if (equal b 'f) 'f
      (if (equal c 'f) 'f
        (if (and (equal a 't) (equal b 't) (equal c 't)) 't
          'x)))))

(defn m-nand3 (a b c)
  (m-not (m-and3 a b c)))

(defn inv ()
  '(combinational  ;type
    (a)            ;inputs
    ((m-not a))    ;outputs
    (inertial)     ;modes
    ((2000))))

(defn nand ()
  '(combinational  ;type
    (a b)          ;inputs
    ((m-nand a b)) ;outputs
    (inertial)     ;modes
    ((2000))))     ;delays
```

```
(defn nand3 ()
  '(combinational    ;type
    (a b c)          ;inputs
    ((m-nand a b c)) ;outputs
    (inertial)       ;modes
    ((2000))))       ;delays


;;*********************************************************************
;;                      SEQUENTIAL MODULES
;;*********************************************************************

;;A sequential module has (along with INPUTS, OUTPUTS, MODES, and DELAYS)
;;six additional components:

(defn trigger (mod)
  ;either POSITIVE-EDGE or NEGATIVE-EDGE
  (cadddddr mod))

(defn locals (mod)
  ;a list of litatoms (internal state variables) from which output forms
  ;are constructed (rather than from input variables)
  (caddddddr mod))

(disable locals)

(defn state (mod)
  ;a list of forms (for computing local values), which may involve locals
  ;and input variables
  (cadddddddr mod))

(disable state)

(defn period (mod)
  ;a number
  (caddddddddr mod))

(disable period)

(defn setups (mod)
  ;a list of numbers
  (cadddddddddr mod))
```

```
(disable setups)

(defn holds (mod)
  ;a list of numbers
  (cadddddddddddr mod))

(disable holds)

;;A positive-edge-triggered device:

(defn d-flip-flop ()
  '(sequential          ;type
    (clk d)             ;inputs
    (q (m-not q))       ;outputs
    (inertial inertial) ;modes
    ((4000 . 6000)      ;delays
     (4000 . 6000))
    positive-edge       ;trigger
    (q)                 ;locals
    (d)                 ;state
    12000               ;period
    (6000 4000)         ;setups
    (6000 4000)))       ;holds

(defn ncopies (n x)
  (if (zerop n)
      ()
      (cons x (ncopies (sub1 n) x))))

(defn kill-state (mod)
  (ncopies (length (locals mod)) 'x))

(defn next-state (state inputs mod)
  (eval$ 'list
         (state mod)
         (append (pairlist (locals mod) state)
                 (pairlist (cdr (inputs mod)) inputs))))

(defn check-clock-setup-or-hold (w time)
  (and (equal (caadr w) (m-not (caar w)))
       (tleq (tplus (cdadr w) time) (cdar w))))

(defn check-data-setups (inp time setups)
  (if (listp inp)
```

```
      (and (not (equal (last-value (car inp)) 'x))
           (tleq (tplus (cdaar inp) (car setups)) time)
           (check-data-setups (cdr inp) time (cdr setups)))
    t))

(defn check-period (w period)
  (and (equal (caaddr w) (caar w))
       (tleq (tplus (cdaddr w) period) (cdar w))))

(defn check-data-holds (inp edge time holds)
  (if (listp inp)
      (and (or (not (new-value-p (car inp) time))
               (tleq (tplus edge (car holds)) time))
           (check-data-holds (cdr inp) edge time (cdr holds)))
    t))

(defn last-time (p)
  (if (listp p)
      (if (tlessp (last-time (cdr p)) (cdaar p))
          (cdaar p)
        (last-time (cdr p)))
    (zero-time)))

(defn strip-events (p time)
  (if (listp p)
      (if (equal time (cdaar p))
          (cons (cdar p) (strip-events (cdr p) time))
        (cons (car p) (strip-events (cdr p) time)))
    p))

(prove-lemma leq-count-strip-events (rewrite)
  (not (lessp (count p) (count (strip-events p time)))))

(prove-lemma lessp-count-strip-events (rewrite)
  (implies (and (packetp p n) (not (equal (last-time p) (zero-time))))
           (lessp (count (strip-events p (last-time p)))
                  (count p))))

(disable strip-events)

(disable last-time)

(defn compute-state (mod inp trigger)
  (if (packetp inp (length inp))
```

```
      (let ((time (last-time inp)))
        (if (equal time (zero-time))
            (kill-state mod)
          (if (equal (cdar (car inp)) time)
              (if (equal (caar (car inp)) trigger)
                  (if (and (check-clock-setup-or-hold
                             (car inp) (car (setups mod)))
                           (check-data-setups
                             (cdr inp) time (cdr (setups mod)))
                           (check-period (car inp) (period mod)))
                      (next-state (compute-state mod
                                                 (strip-events inp time)
                                                 trigger)
                                  (last-values (cdr inp))
                                  mod)
                    (kill-state mod))
                (if (and (equal (caar (car inp)) (m-not trigger))
                         (check-clock-setup-or-hold
                           (car inp) (car (holds mod))))
                    (compute-state mod (strip-events inp time) trigger)
                  (kill-state mod)))
            (if (and (equal (cdar (car inp)) trigger)
                     (not (check-data-holds
                            (cdr inp) (cdar (car inp))
                            time (cdr (holds mod)))))
                (kill-state mod)
              (compute-state mod (strip-events inp time) trigger)))))
    f)
  ((lessp (count inp))))

(enable strip-events)

(enable last-time)

(defn sequential-state (mod inp)
  (case (trigger mod)
    (positive-edge (compute-state mod inp 't))
    (negative-edge (compute-state mod inp 'f))
    (otherwise (compute-state mod inp 't))))

(defn sequential-step (mod inp outp time)
  (post-events
    outp
    (eval$ 'list
```

```
          (outputs mod)
          (pairlist (locals mod)
                    (sequential-state mod (packet-history inp time))))
    time
    (modes mod)
    (delays mod)))

(defn behavioral-step (mod inp outp time)
  (case (type mod)
    (combinational (combinational-step mod inp outp time))
    (sequential (sequential-step mod inp outp time))
    (otherwise f)))


;;*****************************************************************
;;                      STRUCTURAL MODULES
;;*****************************************************************

;;Structural modules are built recursively out of submodules.  A
;;structural module has 5 components:

;(defn inputs (mod)
;  ;a list of litatoms
;  (cadr mod))

;(defn outputs (mod)
;  ;a list of litatoms
;  (caddr mod))

(defn submodules (mod)
  ;a list of modules
  (cadddr mod))

(disable submodules)

(defn subinputs (mod)
  ;a list of lists of litatoms
  (caddddr mod))

(disable subinputs)

(defn suboutputs (mod)
  ;a list of lists of litatoms
  (cadddddr mod))
```

```
(disable suboutputs)

(defn unionl (l)
  (if (listp l)
      (union (car l) (unionl (cdr l)))
    ()))

(defn signals (mod)
  (unionl (cons (inputs mod) (suboutputs mod))))

(defn lookup (key keys list)
  (if (listp keys)
      (if (equal key (car keys))
          (car list)
        (lookup key (cdr keys) (cdr list)))
    f))

(defn find-list (key lists)
  (if (listp lists)
      (if (member key (car lists))
          (car lists)
        (find-list key (cdr lists)))
    f))

(defn find-outputs (out mod)
  (find-list out (suboutputs mod)))

(defn lookup-list (key keys list)
  (if (listp keys)
      (if (member key (car keys))
          (car list)
        (lookup-list key (cdr keys) (cdr list)))
    f))

(defn find-submodule (out mod)
  (lookup-list out (suboutputs mod) (submodules mod)))

(defn find-inputs (out mod)
  (lookup-list out (suboutputs mod) (subinputs mod)))

(defn find-delay (out mod)
  (lookup out (find-outputs out mod) (delays (find-submodule out mod))))
```

```
(defn find-mode (out mod)
  (lookup out (find-outputs out mod) (modes (find-submodule out mod))))

;;The following macro is given for convenience in defining structural
;;modules:

(defmacro defcircuit (name inputs outputs &rest occurrences)
  `(defn ,name ()
     `(structural ,',inputs ,',outputs
                  ,,`(list ,@(mapcar #'first occurrences))
                  ,',(mapcar #'second occurrences)
                  ,',(mapcar #'third occurrences))))

;;As an example, we build a D-flip-flop out of nand gates:

(defcircuit d-with-nands
  (clk d) ;inputs
  (q qn)  ;outputs
  ((nand) (b2 b1) (a1))
  ((nand) (a1 clk) (b1))
  ((nand3) (b1 clk b2) (a2))
  ((nand) (a2 d) (b2))
  ((nand) (b1 qn) (q))
  ((nand) (q a2) (qn)))

;;We define two predicates that must be satisfied by any structural
;;module.  The first of these, SYNTAX-OK, checks that all list have
;;appropriate lengths, etc.:

(defn match-inputs (subins subs)
  (if (listp subs)
      (and (listp subins)
           (equal (length (car subins)) (number-of-inputs (car subs)))
           (match-inputs (cdr subins) (cdr subs)))
    t))

(defn match-outputs (subouts subs)
  (if (listp subs)
      (and (equal (length (car subouts)) (number-of-outputs (car subs)))
           (match-outputs (cdr subouts) (cdr subs)))
    t))

(defn appears (x l)
  (if (listp l)
```

```
      (or (member x (car l))
          (appears x (cdr l)))
    f))

(defn all-appear (l m)
  (if (listp l)
      (and (appears (car l) m)
           (all-appear (cdr l) m))
    t))

(defn lists-all-appear (ls m)
  (if (listp ls)
      (and (all-appear (car ls) m)
           (lists-all-appear (cdr ls) m))
    t))

(defn none-appear (l m)
  (if (listp l)
      (and (not (appears (car l) m))
           (none-appear (cdr l) m))
    t))

(defn distinct-symbols (l)
  (if (listp l)
      (and (litatom (car l))
           (not (member (car l) (cdr l)))
           (distinct-symbols (cdr l)))
    t))

(defn all-distinct-symbols (ls)
  (if (listp ls)
      (and (distinct-symbols (car ls))
           (none-appear (car ls) (cdr ls))
           (all-distinct-symbols (cdr ls)))
    t))

(defn syntax-ok (mod)
  (and (equal (length (subinputs mod)) (length (submodules mod)))
       (match-inputs (subinputs mod) (submodules mod))
       (equal (length (suboutputs mod)) (length (submodules mod)))
       (match-outputs (suboutputs mod) (submodules mod))
       (all-appear (outputs mod) (suboutputs mod))
       (lists-all-appear
         (subinputs mod) (cons (inputs mod) (suboutputs mod)))
```

```
       (all-distinct-symbols (cons (inputs mod) (suboutputs mod)))))

;;The other predicate that must be satisfied by any structural module,
;;DELTA-ACYCLIC, checks for cyclic 0-delay paths.  It is defined in
;;terms of an important auxiliary function, DLEVEL$:

(defn delete (x l)
  (if (listp l)
      (if (equal x (car l))
          (cdr l)
        (cons (car l) (delete x (cdr l))))
    l))

(defn subbagp (l m)
  (if (listp l)
      (and (member (car l) m)
           (subbagp (cdr l) (delete (car l) m)))
    t))

(defn subsetp (l m)
  (if (listp l)
      (and (member (car l) m)
           (subsetp (cdr l) m))
    t))

(prove-lemma length-delete (rewrite)
  (implies (member x l)
           (equal (length (delete x l))
                  (sub1 (length l)))))

(prove-lemma member-delete (rewrite)
  (implies (and (member x l)
                (not (equal x y)))
           (member x (delete y l))))

(prove-lemma lessp-length-subbagp ()
  (implies (and (subbagp l m)
                (member x m)
                (not (member x l)))
           (lessp (length l) (length m))))

(prove-lemma subsetp-delete (rewrite)
  (implies (and (subsetp l m)
                (not (member x l)))
```

```
          (subsetp l (delete x m))))

(prove-lemma subsetp-subbagp (rewrite)
  (implies (and (distinct-symbols l)
                (subsetp l m))
           (subbagp l m))
  ((induct (subbagp l m))))

(prove-lemma lessp-length-subset (rewrite)
  (implies (and (subsetp l m)
                (distinct-symbols l)
                (member x m)
                (not (member x l)))
           (lessp (length l) (length m)))
  ((use (lessp-length-subbagp))))

(defn fmax (x y)
  ;the maximum of x and y, with F treated as infinite
  (if (and x y)
      (max x y)
    f))

(defn select-deltas (delays env)
  (if (listp delays)
      (if (zerop (min-delay (car delays)))
          (cons (car env) (select-deltas (cdr delays) (cdr env)))
        (select-deltas (cdr delays) (cdr env)))
    ()))

(prove-lemma lessp-count-submodules-mod (rewrite)
  (implies (structuralp mod)
           (equal (lessp (count (submodules mod)) (count mod)) t))
  ((enable submodules type)))

;;Suppose IN in a signal of a structural module MOD, ENV is a list of
;;length (NUMBER-OF-OUTPUTS MOD), and BAD is a list of signals of MOD.
;;Assume that for each i < (NUMBER-OF-OUTPUTS MOD), the ith member of
;;ENV is the length of the longest 0-delay path starting at the ith
;;member of (OUTPUTS MOD) and leading outward.  Assume further that
;;there is an infinite (i.e., cyclic) 0-delay path starting at each BAD
;;signal.  Then (DLEVEL$ 0 IN MOD ENV BAD) is the length of the longest
;;0-delay path starting at IN:

(defn lookup-all (x l m)
```

```
  (if (listp l)
      (if (equal x (car l))
          (cons (car m) (lookup-all x (cdr l) (cdr m)))
        (lookup-all x (cdr l) (cdr m)))
    m))

(defn lookup-inputs (x l m)
  (if (listp l)
      (cons (lookup-all x (car l) (inputs (car m)))
            (lookup-inputs x (cdr l) (cdr m)))
    m))

(defn fmaxl (l)
  (if (listp l)
      (fmax (car l) (fmaxl (cdr l)))
    0))

(defn fmaxll (l)
  (if (listp l)
      (fmax (fmaxl (car l)) (fmaxll (cdr l)))
    0))

(defn fadd1 (n)
  (if n (add1 n) f))

(defn fadd1l (l)
  (if (listp l)
      (cons (fadd1 (car l)) (fadd1l (cdr l)))
    ()))

(defn dlevel$ (mode in mod env bad)
  (case mode
    (0 (if (structuralp mod)
           (if  (and (not (member in bad))
                     (equal (length (suboutputs mod))
                            (length (submodules mod)))
                     (member in (signals mod))
                     (distinct-symbols bad)
                     (subsetp bad (signals mod)))
               (fmaxll (cons (lookup-all in (outputs mod) env)
                             (dlevel$
                               3
                               (lookup-inputs in
                                              (subinputs mod)
```

```
                                                (submodules mod))
                                  (submodules mod)
                                (dlevel$ 2 (suboutputs mod) mod env
                                           (cons in bad))
                                ())))
              f)
          (fmaxl (fadd1l (select-deltas (delays mod) env)))))
    (1 (if (listp in)
           (cons (dlevel$ 0 (car in) mod env bad)
                 (dlevel$ 1 (cdr in) mod env bad))
          ()))
    (2 (if (listp in)
           (cons (dlevel$ 1 (car in) mod env bad)
                 (dlevel$ 2 (cdr in) mod env bad))
          ()))
    (3 (if (listp mod)
           (cons (dlevel$ 1 (car in) (car mod) (car env) bad)
                 (dlevel$ 3 (cdr in) (cdr mod) (cdr env) bad))
          ()))
    (otherwise f))
  ((ord-lessp (lex (list (count mod)
                         (difference (length (signals mod)) (length bad))
                         (count in))))))

(defn delta-acyclic (mod)
  ;determines whether there is any cyclic 0-delay path within MOD
  (fmaxll (dlevel$ 2
                   (suboutputs mod)
                   mod
                   (ncopies (number-of-outputs mod) 0)
                   ())))

(defn modulep$ (flag mod)
  (if (equal flag 'list)
      (if (listp mod)
          (and (modulep$ t (car mod))
               (modulep$ 'list (cdr mod)))
        t)
    (case (type mod)
      (structural
        (and (syntax-ok mod)
             (delta-acyclic mod)
             (modulep$ 'list (submodules mod))))
      (combinational
```

```
        (and (equal (length (delays mod)) (length (outputs mod)))
             (equal (length (modes mod)) (length (outputs mod)))))
      (sequential
        (and (equal (length (delays mod)) (length (outputs mod)))
             (equal (length (modes mod)) (length (outputs mod)))
             (equal (length (state mod)) (length (locals mod)))
             (equal (length (holds mod)) (length (inputs mod)))
             (equal (length (setups mod)) (length (inputs mod)))))
      (otherwise f))))

(defn modulep (mod)
  (modulep$ t mod))

;;We shall define a step function for structural modules.  Instead
;;of an output packet, the object on which this function operates
;;(its third argument and its value) is an output "bundle", which
;;consists of a packet corresponding to each behavioral component.

;;First, we extract from a wave bundle the packet corresponding
;;to a module's output signals:

(defn select-wave (key signals packets)
  (if (listp packets)
      (if (member key (car signals))
          (lookup key (car signals) (car packets))
        (select-wave key (cdr signals) (cdr packets)))
    f))

(defn select-packet (keys signals packets)
  (if (listp keys)
      (cons (select-wave (car keys) signals packets)
            (select-packet (cdr keys) signals packets))
    ()))

(defn output-packet$ (flag bundle mod)
  (if (equal flag 'list)
      (if (listp mod)
          (cons (output-packet$ t (car bundle) (car mod))
                (output-packet$ flag (cdr bundle) (cdr mod)))
        ())
    (if (structuralp mod)
        (select-packet
          (outputs mod)
          (suboutputs mod)
```

```
          (output-packet$ 'list bundle (submodules mod)))
      bundle)))

(defn output-packet (bundle module)
  (output-packet$ t bundle module))

;;Next, we extract, from an input packet and a bundle, a list of
;;the input packets to a module's submodules:

(defn input-packet (ins inpacket bundle mod)
  (select-packet
    ins
    (cons (inputs mod) (suboutputs mod))
    (cons inpacket (output-packet$ 'list bundle (submodules mod)))))

(defn input-packets (ins inpacket bundle mod)
  (if (listp ins)
      (cons (input-packet (car ins) inpacket bundle mod)
            (input-packets (cdr ins) inpacket bundle mod))
    ()))

(defn subinput-packets (inpacket bundle mod)
  (input-packets (subinputs mod) inpacket bundle mod))

(defn step$ (flag mod inpacket bundle time)
  (if (equal flag 'list)
      (if (listp mod)
          (cons (step$ t (car mod) (car inpacket) (car bundle) time)
                (step$ 'list (cdr mod) (cdr inpacket) (cdr bundle) time))
        ())
    (if (structuralp mod)
        (step$ 'list
               (submodules mod)
               (subinput-packets inpacket bundle mod)
               bundle
               time)
      (if (some-new-value-p inpacket time)
          (behavioral-step mod inpacket bundle time)
        bundle))))

(defn step (mod inpacket bundle time)
  (step$ t mod inpacket bundle time))

;;*********************************************************************
```

```
;;                          SIMULATION
;;*********************************************************************

;;A simulation of a module is the computation of an output packet
;;produced in response to a given input packet.  We would like to allow
;;both packets to be infinite.  Note that even when the input packet is
;;finite, the output (of a structural module) may never stabilize.
;;Since our implementation does not allow the explicit representation of
;;infinite waveforms, our simulator takes a time argument (in addition
;;to a module and input packet).  The value returned is a wave packet
;;representing the output produced up to that time.

;;The simulator is defined recursively in terms of STEP.  In order to
;;guarantee termination of the recursion, all events are assumed to be
;;scheduled at times whose 2nd (delta) components are uniformly bounded
;;by some number D, which is passed to the simulator as a 4th argument.

;;The valid time that immediately follows a given time is computed as
;;follows:

(defn tinc (time d)
  (if (lessp (cdr time) d)
      (cons (car time) (add1 (cdr time)))
    (cons (add1 (car time)) 0)))

;;We define a function that steps recursively:

(defn walk (mod inpacket bundle start stop d)
  (if (tlessp start stop)
      (walk mod
            inpacket
            (step mod inpacket bundle (tinc start d))
            (tinc start d)
            stop
            d)
    bundle)
  ((ord-lessp (cons (add1 (difference (add1 (car stop)) (car start)))
                    (difference d (cdr start))))))

;;We make no assumptions about the waveforms initially associated with any
;;of the signals produced by MOD.  Thus, we take each of these to be the
;;waveform whose value is everywhere unknown:

(defn null-bundle$ (flag mod)
```

```
  (if (equal flag 'list)
      (if (listp mod)
          (cons (null-bundle$ t (car mod))
                (null-bundle$ 'list (cdr mod)))
        ())
    (if (structuralp mod)
        (null-bundle$ 'list (submodules mod))
      (ncopies (number-of-outputs mod) (list (cons 'x (zero-time)))))))

(defn initialize (mod inp)
  (step mod inp (null-bundle$ t mod) (zero-time)))

(defn sim (mod inp t1 d)
  (packet-history
    (output-packet
      (walk mod inp (initialize mod inp) (zero-time) t1 d) mod)
    t1))


;;*****************************************************************
;;                       DELTA CONSTRAINTS
;;*****************************************************************

;;We require that no event is ever scheduled for a time with delta
;;component exceeding the D argument of WALK.  This imposes a lower
;;bound on D, namely, the maximum of the dlevels of the signals of MOD
;;and its submodules:

(defn dmin$ (flag mod env)
  (if (equal flag 'list)
      (if (listp mod)
          (max (dmin$ t (car mod) (car env))
               (dmin$ 'list (cdr mod) (cdr env)))
        0)
    (if (structuralp mod)
        (max (fmaxll (dlevel$ 2
                              (cons (inputs mod) (suboutputs mod))
                               mod env ()))
             (dmin$ 'list
                    (submodules mod)
                    (dlevel$ 2 (suboutputs mod) mod env ())))
      (max (fmaxl env)
           (fmaxl (fadd1l (select-deltas (delays mod) env)))))))
```

```
(defn dmin (mod env)
  (dmin$ t mod env))

;;Restrictions are similarly imposed on the 2nd and 3rd arguments of
;;WALK:

(defn bounded-delta-p (x d)
  (leq (cdr x) d))

(defn bounded-waveform-p (w d)
  (if (listp w)
      (if (listp (cdr w))
          (and (bounded-waveform-p (cdr w) d)
               (timep (cdar w))
               (bounded-delta-p (cdar w) d)
               (tlessp (cdadr w) (cdar w))
               (not (equal (caadr w) (caar w))))
        (equal (cdar w) (zero-time)))
    f))

(defn bounded-packet-p (p dlist)
  (if (listp dlist)
      (and (listp p)
           (bounded-waveform-p (car p) (car dlist))
           (bounded-packet-p (cdr p) (cdr dlist)))
    (nlistp p)))

(defn differences (d l)
  (if (listp l)
      (cons (difference d (car l))
            (differences d (cdr l)))
    ()))

(defn inpacketp (p mod env d)
  (and (leq (dmin mod env) d)
       (bounded-packet-p
         p (differences d (dlevel$ 1 (inputs mod) mod env ()))))

(defn bundlep$ (flag bun mod env d)
  (if (equal flag 'list)
      (if (listp mod)
          (and (bundlep$ t (car bun) (car mod) (car env) d)
               (bundlep$ 'list (cdr bun) (cdr mod) (cdr env) d))
        (nlistp bun))
```

```
    (if (structuralp mod)
        (bundlep$ 'list
                  bun
                  (submodules mod)
                  (dlevel$ 2 (suboutputs mod) mod env ())
                  d)
      (bounded-packet-p bun (differences d env)))))

(defn bundlep (bun mod env d)
  (bundlep$ t bun mod env d))


;*********************************************************************
                     A FAST SIMULATOR
;*********************************************************************

(defn update-state (mod inp state time trigger)
  (if (equal time (zero-time))
      (kill-state mod)
    (if (new-value-p (car inp) time)
        (if (equal (caar (car inp)) trigger)
            (if (and (check-clock-setup-or-hold
                       (car inp) (car (setups mod)))
                     (check-data-setups
                       (cdr inp) time (cdr (setups mod)))
                     (check-period (car inp) (period mod)))
                (next-state state (last-values (cdr inp)) mod)
              (kill-state mod))
          (if (and (equal (caar (car inp)) (m-not trigger))
                   (check-clock-setup-or-hold
                     (car inp) (car (holds mod))))
              state
            (kill-state mod)))
      (if (and (equal (cdar (car inp)) trigger)
               (not (check-data-holds
                      (cdr inp) (cdar (car inp))
                      time (cdr (holds mod)))))
          (kill-state mod)
        state)))
  ((lessp (count inp))))

(defn fast-sequential-step (mod inp bundle time)
  (let ((state (update-state
                 mod
```

```
                (packet-history inp time)
                (cdr bundle)
                time
                (if (equal (trigger mod) 'negative-edge) 'f 't))))
    (cons (post-events
            (car bundle)
            (eval$ 'list (outputs mod) (pairlist (locals mod) state))
            time
            (modes mod)
            (delays mod))
          state)))

(defn fast-behavioral-step (mod inp bundle time)
  (case (type mod)
    (combinational (combinational-step mod inp bundle time))
    (sequential (fast-sequential-step mod inp bundle time))
    (otherwise f)))

(defn fast-output-packet$ (flag bundle mod)
  (if (equal flag 'list)
      (if (listp mod)
          (cons (fast-output-packet$ t (car bundle) (car mod))
                (fast-output-packet$ flag (cdr bundle) (cdr mod)))
        ())
    (if (structuralp mod)
        (select-packet
          (outputs mod)
          (suboutputs mod)
          (fast-output-packet$ 'list bundle (submodules mod)))
      (if (sequentialp mod)
          (car bundle)
        bundle))))

(defn fast-output-packet (bundle module)
  (fast-output-packet$ t bundle module))

;;Next, we extract, from an input packet and a bundle, a list of
;;the input packets to a module's submodules:

(defn fast-input-packet (ins inpacket bundle mod)
  (select-packet
    ins
    (cons (inputs mod) (suboutputs mod))
    (cons inpacket (fast-output-packet$ 'list bundle (submodules mod)))))
```

```
(defn fast-input-packets (ins inpacket bundle mod)
  (if (listp ins)
      (cons (fast-input-packet (car ins) inpacket bundle mod)
            (fast-input-packets (cdr ins) inpacket bundle mod))
    ()))

(defn fast-subinput-packets (inpacket bundle mod)
  (fast-input-packets (subinputs mod) inpacket bundle mod))

(defn fast-step$ (flag mod inpacket bundle time)
  (if (equal flag 'list)
      (if (listp mod)
          (cons (fast-step$ t (car mod) (car inpacket) (car bundle) time)
                (fast-step$
                  'list (cdr mod) (cdr inpacket) (cdr bundle) time))
        ())
    (if (structuralp mod)
        (fast-step$ 'list
                    (submodules mod)
                    (fast-subinput-packets inpacket bundle mod)
                    bundle
                    time)
      (if (some-new-value-p inpacket time)
          (fast-behavioral-step mod inpacket bundle time)
        bundle))))

(defn fast-step (mod inpacket bundle time)
  (fast-step$ t mod inpacket bundle time))

(defn next-wave-event (wave t0)
  (if (listp wave)
      (if (tlessp t0 (cdar wave))
          (if (tlessp t0 (cdadr wave))
              (next-wave-event (cdr wave) t0)
            (cdar wave))
        f)
    f))

(defn ftmin (t1 t2)
  (if t1
      (if (and t2 (tlessp t2 t1)) t2 t1)
    t2))
```

```
(defn next-packet-event (p t0)
  (if (listp p)
      (ftmin (next-wave-event (car p) t0)
             (next-packet-event (cdr p) t0))
    f))

(defn next-bundle-event$ (flag bun mod t0)
  (if (equal flag 'list)
      (if (listp mod)
          (ftmin (next-bundle-event$ t (car bun) (car mod) t0)
                 (next-bundle-event$ 'list (cdr bun) (cdr mod) t0))
        f)
    (case (type mod)
      (structural (next-bundle-event$ 'list bun (submodules mod) t0))
      (combinational (next-packet-event bun t0))
      (sequential (next-packet-event (car bun) t0))
      (otherwise f))))

(defn next-event (inp fbun mod t0)
  (ftmin (next-packet-event inp t0)
         (next-bundle-event$ t fbun mod t0)))

(prove-lemma tgreaterp-next-wave-event (rewrite)
  (implies (next-wave-event w t0)
           (tlessp t0 (next-wave-event w t0)))
  ((disable tlessp)))

(prove-lemma tgreaterp-next-packet-event (rewrite)
  (implies (next-packet-event p t0)
           (tlessp t0 (next-packet-event p t0)))
  ((disable tlessp)))

(prove-lemma tgreaterp-next-bundle-event (rewrite)
  (implies (next-bundle-event$ flag bun mod t0)
           (tlessp t0 (next-bundle-event$ flag bun mod t0)))
  ((disable tlessp)))

(prove-lemma tgreaterp-next-event (rewrite)
  (implies (next-event inp bun mod t0)
           (tlessp t0 (next-event inp bun mod t0)))
  ((disable tlessp)))

(prove-lemma fast-walk-lemma (rewrite)
  (implies (and (tlessp t0 tnext)
```

```
                (tleq tnext t1)
                (bounded-delta-p tnext d))
           (lex-lessp (list (difference (add1 (car t1)) (car tnext))
                            (difference d (cdr tnext)))
                      (list (difference (add1 (car t1)) (car t0))
                            (difference d (cdr t0))))))

(disable tlessp)

(disable next-event)

(disable lex-lessp)

(disable difference)

(defn fast-walk (mod inpacket fbundle start stop d)
  (let ((tnext (next-event inpacket fbundle mod start)))
    (if tnext
        (if (bounded-delta-p tnext d)
            (if (tlessp stop tnext)
                fbundle
              (fast-walk mod
                         inpacket
                         (fast-step mod inpacket fbundle tnext)
                         tnext
                         stop
                         d))
          f)
      fbundle))
  ((ord-lessp (lex (list (difference (add1 (car stop)) (car start))
                         (difference d (cdr start)))))))

(enable tlessp)

(enable next-event)

(enable lex-lessp)

(enable difference)

(defn null-fbundle$ (flag mod)
  (if (equal flag 'list)
      (if (listp mod)
          (cons (null-fbundle$ t (car mod))
```

```
                (null-fbundle$ 'list (cdr mod)))
        ())
    (if (structuralp mod)
        (null-fbundle$ 'list (submodules mod))
      (if (combinationalp mod)
          (ncopies (number-of-outputs mod) (list (cons 'x (zero-time))))
        (cons (ncopies (number-of-outputs mod)
                       (list (cons 'x (zero-time))))
              (kill-state mod))))))

(defn fast-initialize (mod inp)
  (fast-step mod inp (null-fbundle$ t mod) (zero-time)))

(defn extract-bundle$ (flag fbun mod)
  (if (equal flag 'list)
      (if (listp mod)
          (cons (extract-bundle$ t (car fbun) (car mod))
                (extract-bundle$ 'list (cdr fbun) (cdr mod)))
        ())
    (if (structuralp mod)
        (extract-bundle$ 'list fbun (submodules mod))
      (if (combinationalp mod)
          fbun
        (car fbun)))))

(defn extract-bundle (fbun mod)
  (extract-bundle$ t fbun mod))

(defn fast-sim (mod inp t1 d)
  (packet-history
    (output-packet
      (extract-bundle
        (fast-walk mod inp (fast-initialize mod inp) (zero-time) t1 d)
        mod)
      mod)
    t1))


;*********************************************************************
                        FAST-DLEVEL$
;*********************************************************************

(defn pushl (list stack)
  (if (listp list)
```

```
      (cons (cons (car list) stack)
            (pushl (cdr list) stack))
    ()))

(defn good-list (mod bad)
  (if (listp mod)
      (cons (difference (length (signals (car mod))) (length (car bad)))
            (good-list (cdr mod) (cdr bad)))
    ()))

(defn struct-depth$ (flag mod)
  (if (equal flag 'list)
      (if (listp mod)
          (max (struct-depth$ t (car mod))
               (struct-depth$ 'list (cdr mod)))
        1)
    (if (structuralp mod)
        (add1 (struct-depth$ 'list (submodules mod)))
      1)))

(defn struct-depth (mod)
  (struct-depth$ t mod))

(defn zero-pad (l n)
  (if (lessp n (length l))
      (zero-pad (cdr l) n)
    (if (lessp (length l) n)
        (cons 0 (zero-pad l (sub1 n)))
      l))
  ((lessp (plus (count l) n))))

(prove-lemma length-zero-pad (rewrite)
  (equal (length (zero-pad l n)) (fix n)))

(defn lex-max (x y)
  (if (lex-lessp x y) y x))

(defn reverse (x)
  (if (listp x)
      (append (reverse (cdr x)) (list (car x)))
    ()))

(defn good-measure-1 (in mod bad n)
  (reverse (append (list (count in) (count mod))
```

```
                  (zero-pad (good-list mod bad) n))))

(defn good-measure-2 (mode in mod bad n)
  (if (equal mode 3)
      (if (listp mod)
          (lex-max (good-measure-1 (car in) (car mod) bad n)
                   (good-measure-2 3 (cdr in) (cdr mod) bad n))
        (ncopies (plus 2 n) 0))
    (good-measure-1 in mod bad n)))

(defn good-measure (mode in mod bad n)
  (if (equal mode 3)
      (lex (append (good-measure-2 mode in mod bad n) (list (count mod))))
    (lex (append (good-measure-2 mode in mod bad n) (list 0)))))

(prove-lemma length-reverse (rewrite)
  (equal (length (reverse l)) (length l)))

(defn tailp (s l k)
  (if (zerop k)
      (equal s l)
    (tailp s (cdr l) (sub1 k))))

(prove-lemma lex-lessp-reverse ()
  (implies (and (lex-lessp (reverse g1) (reverse g2))
                (tailp g1 l1 k)
                (tailp g2 l2 k))
           (lex-lessp (reverse l1) (reverse l2))))

(prove-lemma lex-lessp-append (rewrite)
  (implies (lessp a b)
           (lex-lessp (append l (cons a ()))
                      (append l (cons b ()))))
  ((induct (length l))))

(prove-lemma lex-lessp-reverse-good-list (rewrite)
  (implies (and (listp mod)
                (equal (type (car mod)) 'structural)
                (not (member in (car bad)))
                (member in (signals (car mod)))
                (distinct-symbols (car bad))
                (subsetp (car bad) (signals (car mod))))
           (lex-lessp
             (reverse
```

```
              (good-list
                mod (cons (cons in (car bad)) (cdr bad))))
            (reverse (good-list mod bad)))))

(prove-lemma tailp-zero-pad ()
  (implies (leq (length g) n)
           (tailp g (zero-pad g n) (difference n (length g)))))

(prove-lemma tailp-cdr ()
  (implies (and (tailp g l k) (listp g))
           (tailp (cdr g) l (add1 k))))

(prove-lemma length-good-list (rewrite)
  (equal (length (good-list mod bad)) (length mod)))

(prove-lemma difference-sub1 (rewrite)
  (equal (difference (sub1 x) y)
         (sub1 (difference x y))))

(prove-lemma lex-lessp-reverse-zero-pad (rewrite)
  (implies (and (listp mod)
                (lessp (length mod) n)
                (equal (type (car mod)) 'structural)
                (not (member in (car bad)))
                (member in (signals (car mod)))
                (distinct-symbols (car bad))
                (subsetp (car bad) (signals (car mod))))
           (lex-lessp
             (reverse
               (zero-pad (good-list (cons sub mod)
                                    (cons ()
                                          (cons (cons in (car bad))
                                                (cdr bad))))
                         n))
             (reverse (zero-pad (good-list mod bad) n))))
  ((use (tailp-zero-pad
          (g (good-list (cons sub mod)
                        (cons () (cons (cons in (car bad))
                                       (cdr bad))))))
        (tailp-cdr
          (g (good-list
               (cons sub mod)
               (cons () (cons (cons in (car bad)) (cdr bad)))))
          (l (zero-pad
```

```
              (good-list
                (cons sub mod)
                (cons () (cons (cons in (car bad)) (cdr bad))))
              n))
          (k (sub1 (difference n (length mod)))))
        (lex-lessp-reverse
          (g1 (good-list mod (cons (cons in (car bad)) (cdr bad))))
          (l1 (zero-pad
                (good-list
                  (cons sub mod)
                  (cons () (cons (cons in (car bad)) (cdr bad))))
                n))
          (k (difference n (length mod)))
          (g2 (good-list mod bad))
          (l2 (zero-pad (good-list mod bad) n)))
        (tailp-zero-pad (g (good-list mod bad))))
   (disable zero-pad signals tailp)))

(prove-lemma ord-lessp-good-measure-0 (rewrite)
  (implies (and (listp mod)
                (lessp (length mod) n)
                (equal (type (car mod)) 'structural)
                (not (member in (car bad)))
                (member in (signals (car mod)))
                (distinct-symbols (car bad))
                (subsetp (car bad) (signals (car mod))))
           (lex-lessp
             (good-measure-1
               ins
               (cons sub mod)
               (cons () (cons (cons in (car bad)) (cdr bad))) n)
             (good-measure-2 0 in mod bad n)))
  ((use (lex-lessp-reverse
          (g1 (zero-pad
                (good-list
                  (cons sub mod)
                  (cons () (cons (cons in (car bad)) (cdr bad))))
                n))
          (l1 (append (list (count ins) (count (cons sub mod)))
                      (zero-pad
                        (good-list
                          (cons sub mod)
                          (cons () (cons (cons in (car bad))
                                         (cdr bad))))
```

```
                        n)))
          (g2 (zero-pad (good-list mod bad) n))
          (l2 (append (list (count in) (count mod))
                      (zero-pad (good-list mod bad) n)))
          (k 2)))
   (disable zero-pad signals good-list reverse)))

(prove-lemma good-measure-2-open-1 (rewrite)
  (implies (listp mod)
           (equal (good-measure-2 3 in mod bad n)
                  (lex-max
                    (good-measure-1 (car in) (car mod) bad n)
                    (good-measure-2 3 (cdr in) (cdr mod) bad n)))))

(disable good-measure-2-open-1)

(prove-lemma good-measure-2-open-2 (rewrite)
  (implies (nlistp mod)
           (equal (good-measure-2 3 in mod bad n)
                  (ncopies (plus 2 n) 0))))

(disable good-measure-2-open-2)

(prove-lemma not-ord-lessp-0 (rewrite)
  (implies (equal (length x) (fix k))
           (not (lex-lessp x (ncopies k 0)))))

(prove-lemma lex-lessp-append-a1-a2 ()
  (implies (and (not (lex-lessp a1 a2))
                (lex-lessp b2 b1)
                (equal (length a1) (length a2)))
           (lex-lessp (append a2 b2) (append a1 b1))))

(prove-lemma length-ncopies (rewrite)
  (equal (length (ncopies n x)) (fix n)))

(prove-lemma not-zerop-count-cons ()
  (not (zerop (count (cons (car mod) (cdr mod))))))

(prove-lemma count-listp ()
  (implies (listp mod) (not (zerop (count mod))))
  ((use (not-zerop-count-cons))
   (disable count-cons)))
```

```
(prove-lemma ncopies-plus-n-2 ()
  (equal (append (ncopies n 0) '(0 0))
         (ncopies (plus n 2) 0)))

(prove-lemma assoc-plus ()
  (equal (plus x y) (plus y x)))

(prove-lemma ncopies-plus-2-n ()
  (equal (append (ncopies n 0) '(0 0))
         (ncopies (plus 2 n) 0))
  ((use (ncopies-plus-n-2)
        (assoc-plus (x 2) (y n)))))

(prove-lemma append-append (rewrite)
  (equal (append (append a b) c)
         (append a (append b c))))

(disable append-append)

(prove-lemma lex-leq-0 (rewrite)
  (implies (listp mod)
           (lex-lessp (ncopies (plus 2 n) 0)
                      (good-measure-2 0 in mod bad n)))
  ((use (lex-lessp-append-a1-a2
          (a1 (reverse (zero-pad (good-list mod bad) n)))
          (a2 (ncopies n 0))
          (b1 (list (count mod) (count in)))
          (b2 (list 0 0)))
        (ncopies-plus-2-n)
        (count-listp))
   (enable append-append)))

(prove-lemma lex-lessp-good-measure-3 (rewrite)
  (implies (and (listp mod)
                (lessp (length mod) n)
                (equal (type (car mod)) 'structural)
                (not (member in (car bad)))
                (member in (signals (car mod)))
                (distinct-symbols (car bad))
                (subsetp (car bad) (signals (car mod))))
           (lex-lessp
             (good-measure-2
               3
               ins
```

```
              (pushl subs mod)
              (cons () (cons (cons in (car bad)) (cdr bad)))
              n)
            (good-measure-2 0 in mod bad n)))
  ((disable good-measure-2 good-measure-1 signals ord-lessp count-cons)
   (enable good-measure-2-open-1 good-measure-2-open-2)
   (induct (good-list subs ins))))

(prove-lemma lex-lessp-append-2 ()
  (implies (lex-lessp a1 a2)
           (lex-lessp (append a1 b1) (append a2 b2))))

(prove-lemma length-append (rewrite)
  (equal (length (append a b))
         (plus (length a) (length b))))

(prove-lemma length-good-measure-2 (rewrite)
  (equal (length (good-measure-2 mode in mod bad n))
         (plus n 2)))

(prove-lemma ord-lessp-good-measure-3 (rewrite)
  (implies (and (listp mod)
                (lessp (length mod) n)
                (equal (type (car mod)) 'structural)
                (not (member in (car bad)))
                (member in (signals (car mod)))
                (distinct-symbols (car bad))
                (subsetp (car bad) (signals (car mod))))
           (ord-lessp
             (good-measure
               3
               ins
               (pushl subs mod)
               (cons () (cons (cons in (car bad)) (cdr bad)))
               n)
             (good-measure 0 in mod bad n)))
  ((disable good-measure-2)
   (use (lex-lessp-append-2
         (a1 (good-measure-2
               3
               ins
               (pushl subs mod)
               (cons () (cons (cons in (car bad)) (cdr bad))) n))
         (a2 (good-measure-2 0 in mod bad n))
```

```
          (b1 (list (count (pushl subs mod))))
          (b2 (list 0))))))

(prove-lemma not-lex-lessp-append (rewrite)
  (implies (and (not (lex-lessp a1 a2))
                (not (lex-lessp b1 b2))
                (equal (length a1) (length a2)))
           (not (lex-lessp (append a1 b1) (append a2 b2)))))

(prove-lemma zero-pad-cons-0 ()
  (implies (lessp (length g) n)
           (equal (zero-pad g n)
                  (zero-pad (cons 0 g) n))))

(prove-lemma lex-lessp-lex-lessp-append (rewrite)
  (implies (and (lex-lessp a b)
                (equal (length c) (length d)))
           (lex-lessp (append a c) (append b d))))

(prove-lemma not-lex-lessp-reverse-zero-pad ()
  (implies (and (not (lex-lessp (reverse g1) (reverse g2)))
                (equal (length g1) (length g2)))
           (not (lex-lessp (reverse (zero-pad g1 n))
                           (reverse (zero-pad g2 n))))))

(prove-lemma zero-pad-cdr (rewrite)
  (implies (and (listp g)
                (lessp n (length g)))
           (equal (zero-pad (cdr g) n)
                  (zero-pad g n))))

(prove-lemma lex-leq-zero-pad-cdr ()
  (implies (listp g)
           (not (lex-lessp (reverse (zero-pad g n))
                           (reverse (zero-pad (cdr g) n)))))
  ((use (not-lex-lessp-reverse-zero-pad
          (g1 g) (g2 (cons 0 (cdr g))))
        (zero-pad-cons-0 (g (cdr g))))))

(prove-lemma append-append-append (rewrite)
  (equal (append (append (append a b) c) d)
         (append a (append b (append c d)))))

(disable append-append-append)
```

```
(prove-lemma ord-lessp-good-measure-1 (rewrite)
  (implies (listp mod)
           (ord-lessp (good-measure 1 ins (cdr mod) (cdr bad) n)
                      (good-measure 0 in mod bad n)))
  ((use (lex-leq-zero-pad-cdr (g (good-list mod bad)))
        (lex-lessp-append-a1-a2
          (a1 (reverse (zero-pad (good-list mod bad) n)))
          (a2 (reverse (zero-pad (cdr (good-list mod bad)) n)))
          (b1 (list (count mod) (count in) 0))
          (b2 (list (count (cdr mod)) (count ins) 0))))
   (enable append-append-append)
   (disable zero-pad signals)))

(prove-lemma ord-lessp-good-measure-i (rewrite)
  (implies (and (listp in)
                (not (equal i 3))
                (not (equal j 3)))
           (ord-lessp (good-measure i (cdr in) mod bad n)
                      (good-measure j in mod bad n)))
  ((use (lex-lessp-append-a1-a2
          (a1 (reverse (cons (count mod)
                             (zero-pad (good-list mod bad) n))))
          (a2 (reverse (cons (count mod)
                             (zero-pad (good-list mod bad) n))))
          (b1 (list (count in) 0))
          (b2 (list (count (cdr in)) 0))))
   (enable append-append)
   (disable good-list)))

(prove-lemma ord-lessp-good-measure-i-car (rewrite)
  (implies (and (listp in)
                (not (equal i 3))
                (not (equal j 3)))
           (ord-lessp (good-measure i (car in) mod bad n)
                      (good-measure j in mod bad n)))
  ((use (lex-lessp-append-a1-a2
          (a1 (reverse (cons (count mod)
                             (zero-pad (good-list mod bad) n))))
          (a2 (reverse (cons (count mod)
                             (zero-pad (good-list mod bad) n))))
          (b1 (list (count in) 0))
          (b2 (list (count (car in)) 0))))
   (enable append-append)
```

```
   (disable good-list)))

(prove-lemma ord-lessp-trans-1 (rewrite)
  (implies (and (ord-lessp b c) (ord-lessp a b))
           (ord-lessp a c)))

(prove-lemma length-append-good-measure-2 ()
  (equal (length (append (reverse (zero-pad (good-list x bad) n))
                         (cons (count x) (cons (count (car in)) '(0)))))
         (length (append (good-measure-2 3 (cdr in) z bad n)
                         (list (add1 (plus (count x) (count z))))))))

(prove-lemma lex-lessp-antisymmetry ()
  (not (and (lex-lessp x y) (lex-lessp y x))))

(prove-lemma not-lex-lessp-good-measure-2-3 ()
  (implies (listp mod)
           (not (lex-lessp (good-measure-2 3 in mod bad n)
                           (good-measure-2 1 (car in) (car mod) bad n))))
  ((use (lex-lessp-antisymmetry
          (x (good-measure-2 3 (cdr in) (cdr mod) bad n))
          (y (good-measure-2 1 (car in) (car mod) bad n))))))

(prove-lemma length-append-good-measure-1 (rewrite)
  (equal (length (append (good-measure-1 (car in) (car mod) bad n)
                         '(0)))
         (length (append (good-measure-2 3 in mod bad n)
                         (list (count mod))))))

(prove-lemma ord-lessp-good-measure-1-3 (rewrite)
  (implies (listp mod)
           (ord-lessp (good-measure 1 (car in) (car mod) bad n)
                      (good-measure 3 in mod bad n)))
  ((use (lex-lessp-append-a1-a2
          (a1 (good-measure-2 3 in mod bad n))
          (a2 (good-measure-2 1 (car in) (car mod) bad n))
          (b1 (list (count mod)))
          (b2 '(0)))
        (not-lex-lessp-good-measure-2-3))
   (disable good-measure-1)))

(prove-lemma length-good-measure-1 (rewrite)
  (equal (length (good-measure-1 in mod bad n))
         (plus n 2)))
```

```
(prove-lemma ord-lessp-good-measure-3-3 (rewrite)
  (implies (listp mod)
           (ord-lessp (good-measure 3 (cdr in) (cdr mod) bad n)
                      (good-measure 3 in mod bad n)))
  ((enable good-measure-2-open-1)
   (use (lex-lessp-append-a1-a2
          (a2 (GOOD-MEASURE-2 3 (CDR IN) (CDR MOD) BAD N))
          (a1 (good-measure-1 (car in) (car mod) bad n))
          (b2 (list (count (cdr mod))))
          (b1 (list (count mod)))))
   (disable good-measure-2 good-measure-1)))

(prove-lemma ordp-good-measure (rewrite)
  (ordinalp (good-measure mode in mod bad n)))

(disable good-measure)

(defn fast-dlevel$ (mode in mod env bad n)
  (case mode
    (0 (if (listp mod)
           (if (and (structuralp (car mod))
                    (lessp (length mod) n))
               (if  (and (not (member in (car bad)))
                         (equal (length (suboutputs (car mod)))
                                (length (submodules (car mod))))
                         (member in (signals (car mod)))
                         (distinct-symbols (car bad))
                         (subsetp (car bad) (signals (car mod))))
                   (fmaxll
                     (cons (if (listp (cdr mod))
                               (fast-dlevel$
                                 1
                                 (lookup-all
                                   in (outputs (car mod)) (car env))
                                 (cdr mod)
                                 (cdr env)
                                 (cdr bad)
                                 n)
                             (lookup-all
                               in (outputs (car mod)) (car env)))
                           (fast-dlevel$
                             3
                             (lookup-inputs
```

```
                                in
                                (subinputs (car mod))
                                (submodules (car mod)))
                              (pushl (submodules (car mod)) mod)
                              (pushl (suboutputs (car mod)) env)
                              (cons ()
                                    (cons (cons in (car bad)) (cdr bad)))
                              n)))
                    f)
                (if (listp (cdr mod))
                    (fmaxl (fadd1l (fast-dlevel$ 1
                                     (select-deltas
                                       (delays (car mod)) (car env))
                                     (cdr mod)
                                     (cdr env)
                                     (cdr bad)
                                     n)))
                  (fmaxl (fadd1l (select-deltas
                                   (delays (car mod)) (car env))))))
            f))
      (1 (if (listp in)
             (cons (fast-dlevel$ 0 (car in) mod env bad n)
                   (fast-dlevel$ 1 (cdr in) mod env bad n))
            ()))
      (2 (if (listp in)
             (cons (fast-dlevel$ 1 (car in) mod env bad n)
                   (fast-dlevel$ 2 (cdr in) mod env bad n))
            ()))
      (3 (if (listp mod)
             (cons (fast-dlevel$ 1 (car in) (car mod) (car env) bad n)
                   (fast-dlevel$ 3 (cdr in) (cdr mod) (cdr env) bad n))
            ()))
      (otherwise f))
    ((ord-lessp (good-measure mode in mod bad n))))

(defn fast-delta-acyclic (mod)
  ;determines whether there is any cyclic 0-delay path within MOD
  (fmaxll (fast-dlevel$ 2
            (suboutputs mod)
            (list mod)
            (list (ncopies (number-of-outputs mod) 0))
            (list ())
            (struct-depth mod))))
```

```
;;*******************************************************************
;;                        MODULE REDUCTION
;;*******************************************************************

;;Functions that traverse structural modules will have an argument that
;;represents a bound on the length of the path to be traversed, in order
;;to establish termination.  For this purpose, we define a function that
;;computes the length of the longest path through combinational
;;components of a structure:

(defn slevel$ (flag out mod bad)
  ;(SLEVEL$ T OUT MOD ()) is the length of the longest path through
  ;combinational components to OUT.  MOD is assumed to be a flat
  ;structure.
  (if (equal flag 'list)
      (if (listp out)
          (fmax (slevel$ t (car out) mod bad)
                (slevel$ 'list (cdr out) mod bad))
        0)
    (if (or (member out (inputs mod))
            (sequentialp (find-submodule out mod)))
        0
      (if (and (not (member out bad))
               (distinct-symbols bad)
               (member out (signals mod))
               (subsetp bad (signals mod)))
          (fadd1 (slevel$ 'list (find-inputs out mod) mod (cons out bad)))
        f)))
  ((ord-lessp (lex (list (difference (length (signals mod)) (length bad))
                         (count out))))))

(defn sdepth (mod)
  ;the maximum length of all paths through combinational components
  (slevel$ 'list (signals mod) mod ()))

;;Output delays are computed by tracing backwards to sequential
;;outputs and global inputs:

(defn max-delay-to-signal$ (flag out mod d)
  (if (equal flag 'list)
      (if (listp out)
          (cons (max-delay-to-signal$ t (car out) mod d)
                (max-delay-to-signal$ 'list (cdr out) mod d))
```

```
        ())
    (if (member out (inputs mod))
        0
      (if (sequentialp (find-submodule out mod))
          (max-delay (find-delay out mod))
        (if (zerop d)
            f
          (plus (fmaxl (max-delay-to-signal$
                         'list (find-inputs out mod) mod (sub1 d)))
                (max-delay (find-delay out mod)))))))
  ((ord-lessp (lex (list d (count out))))))

(defn fmin (x y)
  (if x
      (if y
          (if (lessp x y)
              x
            y)
        x)
    y))

(defn fminl (l)
  (if (listp l)
      (fmin (car l) (fminl (cdr l)))
    f))

(defn min-delay-to-signal$ (flag out mod d)
  (if (equal flag 'list)
      (if (listp out)
          (cons (min-delay-to-signal$ t (car out) mod d)
                (min-delay-to-signal$ 'list (cdr out) mod d))
        ())
    (if (member out (inputs mod))
        0
      (if (sequentialp (find-submodule out mod))
          (min-delay (find-delay out mod))
        (if (zerop d)
            f
          (plus (fminl (min-delay-to-signal$
                         'list (find-inputs out mod) mod (sub1 d)))
                (min-delay (find-delay out mod)))))))
  ((ord-lessp (lex (list d (count out))))))

(defn collect-delays (mod d)
```

```
  (pairlist (min-delay-to-signal$ 'list (outputs mod) mod d)
            (max-delay-to-signal$ 'list (outputs mod) mod d)))


;;The delay mode of an output is nondeterministic unless it is
;;generated directly by a sequential component, in which case it
;;inherits its mode from that component:

(defn collect-all-modes (outs mod)
  (if (listp outs)
      (cons (if (sequentialp (find-submodule (car outs) mod))
                (find-mode (car outs) mod)
              'nondeterministic)
            (collect-all-modes (cdr outs) mod))
    ()))

(defn collect-modes (mod)
  (collect-all-modes (outputs mod) mod))


;;Output forms are constructed by tracing back to locals and global
;inputs:

(defn subst$ (flag vals vars form)
  (if (equal flag 'list)
      (if (listp form)
          (cons (subst$ t vals vars (car form))
                (subst$ 'list vals vars (cdr form)))
        ())
    (if (member form vars)
        (lookup form vars vals)
      (if (nlistp form)
          form
        (if (equal (car form) 'quote)
            form
          (cons (car form) (subst$ 'list vals vars (cdr form))))))))

(defn subst (vals vars form)
  (subst$ t vals vars form))

(defn signal-form$ (flag out mod d)
  ;If D is at least the slevel of OUT, then (SIGNAL-FORM$ T OUT MOD D)
  ;is an expression for the signal OUT in terms of the inputs of MOD and
  ;the locals of its sequential components
```

```
  (if (equal flag 'list)
      (if (listp out)
          (cons (signal-form$ t (car out) mod d)
                (signal-form$ 'list (cdr out) mod d))
        ())
    (if (member out (inputs mod))
        out
      (if (sequentialp (find-submodule out mod))
          (lookup out
                  (find-outputs out mod)
                  (outputs (find-submodule out mod)))
        (if (zerop d)
            f
          (subst (signal-form$ 'list (find-inputs out mod) mod (sub1 d))
                 (inputs (find-submodule out mod))
                 (lookup out
                         (find-outputs out mod)
                         (outputs (find-submodule out mod))))))))
  ((ord-lessp (lex (list d (count out))))))

(defn signal-forms (out mod d)
  (signal-form$ 'list out mod d))

(defn collect-outputs (mod d)
  (signal-forms (outputs mod) mod d))


;;If a structure is acyclic and has only combinational components,
;;then it reduces to a combinational module:

(defn comb-reduce (mod)
  (let ((d (sdepth mod)))
    (if d
        (list 'combinational
              (inputs mod)
              (collect-outputs mod d)
              (collect-modes mod)
              (collect-delays mod d))
      f)))


;;The reduction of a sequential structure requires renaming of signals
;;and locals in order to ensure that the locals and inputs of the
;;resulting module are distinct:
```

```
(prove-lemma lessp-quotient (rewrite)
  (implies (leq 10 n)
           (lessp (quotient n 10) n)))

(prove-lemma lessp-remainder (rewrite)
  (implies (leq 10 n)
           (lessp (remainder n 10) n)))

(defn number-codes (n)
  (if (lessp n 10)
      (cons (plus n 48) 0)
    (append (number-codes (quotient n 10))
            (number-codes (remainder n 10)))))

(defn append-number (a n)
  (pack (append (unpack a) (cons 45 (number-codes n)))))

(defn append-number-in-list (l n)
  (if (listp l)
      (cons (append-number (car l) n)
            (append-number-in-list (cdr l) n))
    ()))

(defn append-number-in-lists (l n)
  (if (listp l)
      (cons (append-number-in-list (car l) n)
            (append-number-in-lists (cdr l) n))
    ()))

(defn append-number-in-term$ (flag term vars n)
  (if (equal flag 'list)
      (if (listp term)
          (cons (append-number-in-term$ t (car term) vars n)
                (append-number-in-term$ 'list (cdr term) vars n))
        ())
    (if (listp term)
        (if (equal (car term) 'quote)
            term
          (cons (car term)
                (append-number-in-term$ 'list (cdr term) vars n)))
      (if (member term vars)
          (append-number term n)
        term))))
```

```
(defn append-numbers-in-module (mod n)
  (case (type mod)
    (sequential
      (list 'sequential
            (append-number-in-list (inputs mod) n)
            (append-number-in-term$ 'list (outputs mod) (locals mod) n)
            (modes mod)
            (delays mod)
            (trigger mod)
            (append-number-in-list (locals mod) n)
            (append-number-in-term$
              'list (state mod) (append (inputs mod) (locals mod)) n)
            (period mod)
            (setups mod)
            (holds mod)))
    (combinational
      (list 'combinational
            (append-number-in-list (inputs mod) n)
            (append-number-in-term$ 'list (outputs mod) (inputs mod) n)
            (modes mod)
            (delays mod)))
    (otherwise f)))

(defn append-numbers-in-submodules (mods n)
  (if (listp mods)
      (cons (append-numbers-in-module (car mods) n)
            (append-numbers-in-submodules (cdr mods) (add1 n)))
    ()))

(defn rename-structure (mod)
  (list 'structural
        (append-number-in-list (inputs mod) (length (submodules mod)))
        (append-number-in-list (outputs mod) (length (submodules mod)))
        (append-numbers-in-submodules (submodules mod) 0)
        (append-number-in-lists (subinputs mod) (length (submodules mod)))
        (append-number-in-lists
          (suboutputs mod) (length (submodules mod)))))

;;Setup and hold times of sequential submodules impose constraints on
;;the stability of the structure's signals:

(defn add-max-delays (delays l)
  (if (listp l)
```

```
      (cons (plus (max-delay (car delays)) (car l))
            (add-max-delays (cdr delays) (cdr l)))
    ()))

(defn compute-setup$ (flag in submods subins subouts mod d)
  ;The period for which IN must remain stable prior to a triggering edge
  ;in order not to violate the setup time of any input to a sequential
  ;submodule is given by
  ;(COMPUTE-SETUP$
  ;  T IN (SUBMODULES MOD) (SUBINPUTS MOD) (SUBOUTPUTS MOD) MOD D))
  (if (equal flag 'list)
      (if (listp in)
          (cons (compute-setup$
                  t (car in) submods subins subouts mod d)
                (compute-setup$
                  'list (cdr in) submods subins subouts mod d))
        ())
    (if (listp submods)
        (if (sequentialp (car submods))
            (fmax (fmaxl (lookup-all in (car subins)
                                     (setups (car submods))))
                  (compute-setup$
                    t in (cdr submods) (cdr subins) (cdr subouts) mod d))
          (if (member in (car subins))
              (if (zerop d)
                  f
                (fmax (fmaxl (add-max-delays
                               (delays (car submods))
                               (compute-setup$ 'list
                                               (car subouts)
                                               (submodules mod)
                                               (subinputs mod)
                                               (suboutputs mod)
                                               mod
                                               (sub1 d))))
                      (compute-setup$ t in (cdr submods) (cdr subins)
                                      (cdr subouts) mod d)))
            (compute-setup$
              t in (cdr submods) (cdr subins) (cdr subouts) mod d)))
      0))
  ((ord-lessp (lex (list d (count submods) (count in))))))

(defn compute-setups (ins mod d)
  (compute-setup$
```

```
    'list ins (submodules mod) (subinputs mod) (suboutputs mod) mod d))

(defn collect-setups (mod d)
  (compute-setups (inputs mod) mod d))

(defn subtract-min-delays (delays l)
  (if (listp l)
      (cons (difference (car l) (min-delay (car delays)))
            (subtract-min-delays (cdr delays) (cdr l)))
    ()))

(defn compute-hold$ (flag in submods subins subouts mod d)
  ;The period for which IN must remain stable following a triggering
  ;edge in order not to violate the hold time of any input to a sequential
  ;submodule is given by
  ;(COMPUTE-HOLD$
  ;  T IN (SUBMODULES MOD) (SUBINPUTS MOD) (SUBOUTPUTS MOD) MOD D))
  (if (equal flag 'list)
      (if (listp in)
          (cons (compute-hold$
                  t (car in) submods subins subouts mod d)
                (compute-hold$
                  'list (cdr in) submods subins subouts mod d))
        ())
    (if (listp submods)
        (if (sequentialp (car submods))
            (fmax (fmaxl (lookup-all in (car subins)
                                     (holds (car submods))))
                  (compute-hold$
                    t in (cdr submods) (cdr subins) (cdr subouts) mod d))
          (if (member in (car subins))
              (if (zerop d)
                  f
                (fmax (fmaxl (subtract-min-delays
                               (delays (car submods))
                               (compute-hold$ 'list
                                              (car subouts)
                                              (submodules mod)
                                              (subinputs mod)
                                              (suboutputs mod)
                                              mod
                                              (sub1 d))))
                      (compute-hold$ t in (cdr submods) (cdr subins)
                                     (cdr subouts) mod d)))
```

```
            (compute-hold$
              t in (cdr submods) (cdr subins) (cdr subouts) mod d)))
      0))
  ((ord-lessp (lex (list d (count submods) (count in))))))

(defn compute-holds (ins mod d)
  (compute-hold$
    'list ins (submodules mod) (subinputs mod) (suboutputs mod) mod d))

(defn collect-holds (mod d)
  (compute-holds (inputs mod) mod d))

;;Reduction of a sequential structure requires that (1) the structure is
;;flat, (2) there are no cycles passing only through combinational
;;components, (3) global outputs are functions of state (and not of
;;global inputs), (4) all sequential submodules have the same trigger
;;and are connected to the same clock, and (5) the minimum delays of the
;;outputs of the sequential components are long enough to respect the
;;hold times of the sequential inputs that they feed:

(defn check-holds (holds delays)
  (if (listp holds)
      (and (leq (car holds) (min-delay (car delays)))
           (check-holds (cdr holds) (cdr delays)))
    t))

(defn check-internal (mod submods subins subouts clk trigger d)
  (if (listp submods)
      (and (if (sequentialp (car submods))
               (and (equal (trigger (car submods)) trigger)
                    (equal (caar subins) clk)
                    (check-holds (compute-holds (car subouts) mod d)
                                 (delays (car submods))))
             (combinationalp (car submods)))
           (check-internal
             mod (cdr submods) (cdr subins) (cdr subouts) clk trigger d))
    t))

(defn check-outputs$ (flag out mod d)
  ;Global outputs are required to be functions of state
  (if (equal flag 'list)
      (if (listp out)
          (and (check-outputs$ t (car out) mod d)
               (check-outputs$ 'list (cdr out) mod d))
```

```
        ())
    (if (member out (inputs mod))
        f
      (if (sequentialp (find-submodule out mod))
          t
        (if (zerop d)
            f
          (check-outputs$ 'list (find-inputs out mod) mod (sub1 d))))))
  ((ord-lessp (lex (list d (count out))))))

(defn check-seq-struct (mod trigger d)
  (and (check-outputs$ 'list (outputs mod) mod d)
       (check-internal mod
                       (submodules mod)
                       (subinputs mod)
                       (suboutputs mod)
                       (car (inputs mod))
                       trigger
                       d)))


;;The minimum clock period is bounded by the maximum of the periods of
;;the sequential components.  It also must be long enough to allow
;;internal signals to stabilize in order to respect setup times:

(defn minimum-period$ (submods subouts mod d)
  (if (listp submods)
      (if (sequentialp (car submods))
          (max (max (period (car submods))
                    (fmaxl (add-max-delays
                            (delays (car submods))
                            (compute-setups (car subouts) mod d))))
               (minimum-period$ (cdr submods) (cdr subouts) mod d))
        (minimum-period$ (cdr submods) (cdr subouts) mod d))
    ()))

(defn minimum-period (mod d)
  (minimum-period$ (submodules mod) (suboutputs mod) mod d))


;;State forms are constructed in the same manner as output forms, by
;;tracing back to locals and global inputs:

(defn collect-all-states (subins submods mod d)
```

```
  (if (listp submods)
      (if (sequentialp (car submods))
          (append (subst$ 'list
                          (signal-forms (car subins) mod d)
                          (inputs (car submods))
                          (state (car submods)))
                  (collect-all-states (cdr subins) (cdr submods) mod d))
        (collect-all-states (cdr subins) (cdr submods) mod d))
    ()))

(defn collect-state (mod d)
  (collect-all-states (subinputs mod) (submodules mod) mod d))


;;The locals of the reduced modules are just the union of the locals of
;;all sequential components:

(defn collect-all-locals (submods)
  (if (listp submods)
      (if (sequentialp (car submods))
          (append (locals (car submods))
                  (collect-all-locals (cdr submods)))
        (collect-all-locals (cdr submods)))
    ()))

(defn collect-locals (mod)
  (collect-all-locals (submodules mod)))


;;Before a sequential structure is reduced, its locals and signals are
;;renamed and its admissibility is established:

(defn reduce-renamed-struct (mod trigger d)
  (list 'sequential
        (inputs mod)
        (collect-outputs mod d)
        (collect-modes mod)
        (collect-delays mod d)
        trigger
        (collect-locals mod)
        (collect-state mod d)
        (minimum-period mod d)
        (collect-setups mod d)
        (collect-holds mod d)))
```

```
(defn seq-reduce (mod trigger)
  (let ((d (sdepth mod)))
    (if (and d (check-seq-struct mod trigger d))
        (reduce-renamed-struct (rename-structure mod) trigger d)
      f)))


;;A structure is reduced after searching for sequential components:

(defn determine-type-of-reduction (mod submods)
  (if (listp submods)
      (case (type (car submods))
        (combinational (determine-type-of-reduction mod (cdr submods)))
        (sequential (seq-reduce mod (trigger (car submods))))
        (otherwise f))
    (comb-reduce mod)))

(defn reduce-structure (mod)
  (determine-type-of-reduction mod (submodules mod)))

(defn reduce-module$ (flag mod)
  (if (equal flag 'list)
      (if (listp mod)
          (cons (reduce-module$ t (car mod))
                (reduce-module$ 'list (cdr mod)))
        ())
    (if (structuralp mod)
        (reduce-structure (list 'structural
                                (inputs mod)
                                (outputs mod)
                                (reduce-module$ 'list (submodules mod))
                                (subinputs mod)
                                (suboutputs mod)))
      mod)))

(defn reduce-module (mod)
  (reduce-module$ t mod))
```