CHAPTER 3

DETERMINATION OF HAZARD-FREEDOM

In this chapter, we present the core material that allows us to do a fast and efficient hazard determination for each node in the decomposed netlist. We began with a formal definition of hazard freedom. We than follow with formal definitions of acknowledgement and monotonicity hazards. We then present a short discussion concerning the generation of the time Petri net and the state graph. While these two circuit representations are assumed to be part of the design information that we began our work with, an understanding of their creation and interrelation is deemed helpful.

At this point, sufficient background exists to began discussion on the core work of this thesis, that being the determination of hazard-freedom at each node in the decomposed netlist. This amoundts to finding stable states using two seperate but related techniques: untimed (speed-independent) stabilization and timed stabilization. Once stable states are found, this stabilization is propagated throughout the state graph. Finally, an examination of the behaviour of each node is made to determine whether or not that particular node is free of acknowledgement and monotonicity hazards. Each node is then annotated with it's hazard properties and this information is passed on to the matching/covering stage.

The verification method described in our work requires that the primary outputs must cut the circuit. In other words, if all primary outputs are removed from the netlist, the netlist would become acyclic. Intuitively, this means there can be no internal cycles in the netlist. Since the goal of this work is to use this verifier as a hazard checker during technology mapping and the technology mapper that we are developing satisfies this restriction, this seems acceptable. However, in the future, we are interested in generalizing this work to the case where there is internal feedback in the netlist.

3.1 Formal Definitions

In [4], the following theorem giving sufficient conditions for correctness of an asynchronous circuit is presented (reworded here to match the semantics of this thesis), and it is later proven in [2].

Theorem 3.1. *(Sufficient conditions for correctness) Let* $NET = \langle V, E \rangle$ *be a circuit implementing the behavior described by* $TPN = \langle\ W,T,P,F,M_0,s_0,l,u,L\ \rangle$*. The* NET *is a correct implementation of the* TPN *if (1) it is complex gate equivalent to the TPN, and (2) it satisfies the acknowledgment and monotonicity properties.*

Our verification algorithm to check these correctness conditions is shown in Figure 3.1. This algorithm takes as input a TPN representing the possible input behavior and the required output behavior and a netlist, NET, representing the circuit to be checked, and it determines if the circuit is correct. When the circuit is not correct, this algorithm reports the locations of the errors that it finds. This algorithm is described in detail in section**??**.

```
verify(TPN,NET) {
  SG = check_equivalence(TPN,NET)
  find_stable_states(TPN,SG,NET)
  check_acknowledgment(SG,NET)
  check_monotonicity(SG,NET)
}
```

Figure 3.1. Top-level algorithm for verification.

3.2 Checking equivalence

The `check_equivalence` function forms a complex gate equivalent netlist, uses this netlist and the given TPN to derive a SG, and checks if the complex gate netlist provides outputs only at specified times.

The first step is to derive a complex gate netlist in which there are no internal signals. In other words, it derives a netlist that has one gate per primary output signal in which the Boolean function is only in terms of the primary inputs and outputs. The delay of this gate is set to the minimum and maximum delay from any input to the complex gate to the primary output. In our example, the complex gate representation for the netlists shown in Figure 2.3(b) and Figure 2.3(c) is shown in Figure 2.3(e). The delay for this gate is set to $[1, 4]$, since in both cases there exists a minimum delay path of 1 time unit and a maximum path of 4 times units.

Using this complex gate netlist and the given TPN, a SG is found using a timed state space exploration algorithm. If, during the course of this state space exploration, an output is fired prematurely such as in the example described earlier when using the circuit shown in Figure 2.3(b), an error is reported to the user. Also, if an output is expected and the circuit does not provide one, an error is reported. In our example, if we use the function $f_d = AND(a, c)$, after $a+$ and $b+$, a $d+$ would be expected, but the circuit would not produce it. This models a progress condition similar to *completeness with respect to specification* [17] and *strong conformance* [21]. When no errors are detected, `check_equivalence` returns a SG.

One last thing to note is that during the state space exploration to derive this SG, our method checks that the given complex gate circuit is equivalent to the desired one. For example, if the complex gate circuit given had been the one in Figure 2.3(b), after $a+$ fires, the netlist could produce a $d+$ when one is not expected in the TPN. This complex gate equivalence failure would then be reported to the user.

```
hazard_check() {
  Check circuit output timings vs. TPN timings();
  Find number of states, edges, and nodes in the netlist;
  Init color and eval vectors();
  Find start state;
```

```
  Stabilize edges and propagate coloring information();
  Check for hazards();
}


Check circuit output timings vs. TPN timings() {
  for each output in netlist {
    for each input in netlist {
      if (exists_path) {
        if ((path_length_min < CGE_min) || (path_length_max > CGE_max)) {
          print timing violation;
        }
      }
    }
  }
}


Init color and eval vectors() {
  for each state s in S {
    for each node in netlist {
      if (bool_eval[s,n] == 1) {
        colorvec[n] = RISING;
        evalvec[n] = 1;
      }
      else {
        colorvec[n] = FALLING;
        evalvec[n] = 0;
      }
    }
```

```
    for each succussor edge e of s {
      colorvec[e] = colorvec[s];
    }
  }
}


Stabilize edges and propagate coloring information() {
  do timed stabilization();
  do untimed stabilization();
}


do timed stabilization() {
  for each node in netlist {
    stabilize node timing wise();
    modified = true;
    while (modified) {
      modified = propagate coloring info();
    }
  }
}


stabilize node timing wise() {
  va_nodeR = clear visited array for RISING region;
  va_nodeF = clear visited array for FALLING region;
  va_path = clear visited array for path;
  find_abs_minmax_times from any input to this node;
  for each edge where eval[s] != eval[s'] for this output {
```

```
update_zone(TPN,NET,z,t_i) {
  z' = extend(z,t_i)
  found_causal = FALSE
  foreach t_j ∈ z' in reverse order
    if (t_j ∈ ••t_i) then
      if (!found_causal) then
        found_causal = TRUE
        if (L(t_i) ∈ I) then
          z'_ij = u(t_i)
        else
          z'_ij = find_max_delay(NET,L(t_i))
      if (L(t_i) ∈ I) then
        z'_ji = (−1) ∗ l(t_i)
      else
        z'_ji = (−1)∗ find_min_delay(NET,L(t_i))
    else
      z'_ij = ∞
      z'_ji = ∞
  recanonicalize(z')
}

    for each zone in s {
      z' = update_zone();
      do_timed_recursion();
    }
  }
}

do untimed stabilization() {
  while (modified_high_level) {
    modified_high_level = false;
    for each node in netlist {
      modified_lower_level = true;
      while (modified_lower_level) {
```

```
do_timed(TPN,SG,NET,n,s,z,t_i,d,visit,path) {
  visit(s) = TRUE
  path(s) = TRUE
  foreach (s,t_j,s') in δ
    z' = update_zone(TPN,NET,z,t_j)
    if (-1*z'_ij > d) then
      if (not visit(s')) then
        stable(s,s',n) = TRUE
    else if (not path(s') &&
            eval(s,n) == eval(s',n)) then
      stable(s,s',n) = FALSE
      do_timed(TPN,SG,NET,n,s',z',t_i,d,visit,path)
  path(s) = FALSE
}
```

```
        modified_lower_level = false;
        clear visited array;
        modified_lower_level = stabilize node untimed wise();
        if (modified_lower_level) {
          modified_high_level = true;
        }
        modified_prop = true;
        while (modified_prop) {
          modified_prop = propagate coloring info();
          if (modified_prop) {
            modified_high_level = true;
          }
        }
      }
    }
  }
}
```

```
propagate coloring info() {
```

```
  clear visited array to FALSE;
  if (color_next_state()) {
    visted array[node] = TRUE;
    propagate();
  }
}


propagate() {
  for each successor of state {
    if (!visted) {
      if (color_next_state()) {
visited = TRUE;
propagate();
      }
    }
  }
}


color_next_state() {

}


Check for hazards() {

}
```

```
Discussion items:

1)  Why do timed before untimed?
2)  Why untimed should be iterated but not timed?
3)  All the iterations scenarios during untimed stabilization.
4)  Discussion of tree structure vs. multiple outputs with reconvergant fanout.
5)  Discussion of RISING and FALLING regions for keeping track of visited
    information during timed stabiliazation.
6)  Discussion of getting the initial zone and it's relationship to
    causal events.
7)  Extending timed stabilization to level-based events.
8)  'unwinding' the zone to find the real causal event.
9)  Detailed discussion ofconcurrency and how it relates to causal
events.
```

3.3 Finding stable states

After the `check_equivalence` step, our method has shown that the circuit is correct at a complex gate level. By hiding the internal signals before finding the state space, we reduce the potential size of the state space from $O(2^{|I|} * 2^{|O|} * 2^{|N|})$ to $O(2^{|I|} * 2^{|O|})$ representing an exponential reduction in the potential size of the state space. When the number of internal signals is large, as is often the case in real designs, this savings can be quite dramatic. However, hazards on internal nodes can still produce incorrect circuit behavior. Therefore, it is now necessary to check that all internal nodes are hazard-free. This is accomplished by determining internal signal behavior implicitly. In particular, our method annotates each state with stability information about each internal signal. The goal of the `find_stable_states` algorithm is to determine in which states and for which state transitions in the complex gate SG that each internal node is stable. This is accomplished by deriving a predicate `stable`(s,n) for each state $s \in S$ and node $n \in N$ and another predicate stable(s,s',n) for each state transition $(s, t, s') \in \delta$. This stability information can then be used to determine if there are any hazards in the given netlist.

The algorithm to find the stability information is shown in Figure 3.2. The algorithm begins by first determining the predicate eval(s,v) by finding the Boolean evaluation in each state in the SG for each node in the netlist. This can be accomplished by simply fixing the values for each primary input and output in the netlist to the values given in the state and propagating this information through the netlist. From the SG in Figure 2.3(c) and netlist in Figure 2.3(b), eval(1100,e) and eval(1100,d) are determined to both be 1. For node e, the states in the set $\{1100, 1101, 1111, 1110\}$ evaluate to 1 while the remaining states evaluate to 0.

```
find_stable_states(TPN,SG,NET) {
  find eval(s,v) for all s ∈ S  and v ∈ V
  initialize stable(s,n) and stable(s,s',n) to FALSE
    for all n ∈ N, s ∈ S, and (s,t,s') ∈ δ
  stabilize_timed(TPN,SG,NET)
  do {
    propagate(SG)
    modified = stabilize_untimed(SG,NET)
  while modified
}
```

Figure 3.2. Algorithm for finding stable states.

The algorithm next initializes the stability predicates to FALSE to indicate that initially we do not know whether the internal signals are stable or changing in each state and during each state transition. The goal of the rest of the algorithm is to determine stability of the internal signals, whenever possible. In the next subsection, we briefly review untimed stabilization which comes from the work in [4]. In the following subsection, we discuss our new contribution which is timed stabilization. The timed stabilization routine does not need to be iterated, so it is executed first. The untimed stabilization routine may require iteration since stabilizations on one node of the network can influence stabilizations on other nodes.

3.3.1 Untimed stabilization

The objective of stabilization is to show that at some points in the SG, the evaluations of some internal node, n, are certain to be stable. The algorithm to determine untimed stability is shown in Figure 3.3. An internal node is considered untimed stable if a change in evaluation on an internal node is acknowledged on a primary output. In other words, for a state transition (s, t, s'), if the transition t could only have occurred if the internal node n is stable at its Boolean evaluation, then we can say that the transition t has acknowledged that the node n is stable.

```
stabilize_untimed(SG,NET) {
  foreach n ∈ N
    foreach (s,t,s') ∈ δ
      if ((exists_path(NET,n,L(t))) &&
           (must_prop(NET,s,n,L(t)))) then
        stable(s,s',n) = TRUE;
}
```

Figure 3.3. Untimed stabilization algorithm.

To determine if an internal node n is acknowledged to be stable by a state transition (s, t, s'), we first must determine if a path exists from n to the output transition under consideration. It must then be determined using the function `must_prop` if the value at n can propagate through any possible path to the output. This is done by ensuring that all functions in the path from n to the output have non-controlling values on the side inputs. Consider the example netlist in Figure 2.3(b) and the state transition (1100,d+,1101). There exists a path between node e and the output d. In state 1100, node e evaluates to 1. This value at e must propagate to the output because d cannot go high until e has gone high. More succinctly, output d switched from low to high as a direct consequence of node e going high and the side input, c being at 0. Therefore, `stable`(1100,1101,e) is set to TRUE. Next, the `propagate` function is used to propagate this stabilization forward in the state graph until a change in evaluation is encountered. In particular, `stable`(1100, 1101, e) implies the following stability conditions are TRUE: `stable`(1101, e), `stable`(1101, 1111,

e), `stable`(1111, e), `stable`(1111, 1110, e), `stable`(1110, e), and `stable`(1110, 0110, e). The propagation of stability information is halted when it reaches state 0110 because the Boolean evaluation of e in this state changes to 0.

The other transition in the SG that could possibly indicate a stabilization for node e is the state transition (1111, $d-$, 1110). In this case, however, the input c is 1 (a controlling value), prohibiting the propagation of node e to the output d. Thus, no stabilization can be assumed for the falling transition of e. As explained later, this lack of stabilization on the falling transition of e indicates a hazard on node e.

A similar analysis done on the circuit in Figure 2.3(c) shows that the rising transition on node e is acknowledged by $d+$ and the falling transition is acknowledged by $d-$ since b is high (a non-controlling value) when d goes low. Therefore, this circuit is hazard-free under a speed-independent model.

3.3.2 Timed stabilization

When timing information is taken into account, the hazard found for the netlist shown in Figure 2.3(b) may not actually manifest. If this is the case, then node e is hazard-free. This subsection describes our new method to determine stabilization using timing information. Timed stabilization attempts to show further stability in the state graph by calculating the maximum possible time through the network to the node of interest and comparing against the minimum time spent traversing the state graph. When it can be shown that in the worst-case a sufficient amount of time has elapsed, node n can be stabilized.

The algorithm to determine timed stabilization is shown in Figure 3.4. For each node n, the algorithm first measures the longest path delay from any primary input or output to the node n.[1] For our example netlist in Figure 2.3(b), this delay is determined to be 2. Next, the algorithm initializes the `visit` array which is used to let the recursion know when a state has been visited along multiple paths

[1] In order to be safe, the absolute longest path is used and not just the longest path from $L(t_i)$. This must be done because the actual signal that causes n to change evaluation may not be $L(t_i)$ due to differences in path lengths.

when determining stabilization of node n. At this point, the algorithm finds state transitions, (s, t_i, s'), where the Boolean evaluation of n changes.

This indicates locations in the state graph where the node n becomes unstable. The algorithm then takes the zone z associated with state s and updates it to include the transition t_i. The reason this is done rather than taking the zone associated with s' is that t_i may have been pruned from this zone. It is important that t_i is in the zone that we use for timed stabilization as t_i serves as a reference transition as we move forward in the state graph. Finally, the algorithm initializes a `path` array which is used to terminate cycles during the analysis of a path in the SG.

```
stabilize_timed(TPN,SG,NET) {
  foreach (n ∈ N)
    d = find_maximum_delay(NET,n)
    initialize visit(s) for all s ∈ S
    foreach (s,t_i,s') ∈ δ where s=⟨v,z⟩
      if (eval(s,n) ≠ eval(s',n)) then
        z' = update_zone(TPN,NET,z,t_i)
        initialize path(s'') for all s'' ∈ S
        do_timed(TPN,SG,NET,n,s',z',t_i,d,visit,path)
}
```

Figure 3.4. Timed stabilization algorithm.

The `update_zone` algorithm is used to add a new transition to a given zone, and is shown in Figure 3.5. The first step is to extend the zone to include a new row and column for the new transition t_i. Next, it searches the zone starting with the transitions that have been added most recently for transitions that enable t_i (i.e., $t_j \in \bullet \bullet t_i$). The first such transition that it finds is the causal transition for t_i. The upper bound of the firing time for t_i should be set in reference to this transition. The upper bound is either taken from the TPN when t_i is a transition on an input wire or it is taken as the maximum delay in the netlist generating t_i when it is a transition on an output wire. For all transitions that enable t_i, a lower bound must be set between t_j and t_i. For all transitions that do not enable t_i, the

```
update_zone(TPN,NET,z,t_i) {
  z' = extend(z,t_i)
  found_causal = FALSE
  foreach t_j ∈ z' in reverse order
    if (t_j ∈ ••t_i) then
      if (!found_causal) then
        found_causal = TRUE
        if (L(t_i) ∈ I) then
          z'_ij = u(t_i)
        else
          z'_ij = find_max_delay(NET,L(t_i))
      if (L(t_i) ∈ I) then
        z'_ji = (-1) * l(t_i)
      else
        z'_ji = (-1)* find_min_delay(NET,L(t_i))
    else
      z'_ij = ∞
      z'_ji = ∞
  recanonicalize(z')
}
```

Figure 3.5. Algorithm to update the zone.

timing relationships are initially set to be unbounded. At this point, the zone is recanonicalized using Floyd's all-pairs shortest path algorithm to tighten any loose inequalities. As an example, the zone found for the state 1110 in our example is shown in Figure 3.6(a). The new zone after adding the transition $a-$ is shown in Figure 3.6(b).

The `do_timed` algorithm shown in Figure 3.7 is used to recursively explore the SG, attempting to accumulate sufficient time to stabilize a given node n before reaching a termination condition. This algorithm first marks the current state s as visited in the `visit` and `path` arrays described earlier. Next, it considers each state transition (s, t_j, s'). First, it adds the transition, t_j, to the zone. Next, it checks the zone to determine if enough time has accumulated from the reference transition t_i to the new transition t_j such that the node of interest n has certainly stabilized. If it has, it must also check that the state s' has not been visited along a different path. It must be the case that the minimum time upon reaching a state

$$\begin{array}{c|c|} & \tau_{d-} \\ \tau_{d-} & 0 \end{array}$$

(a)

$$\begin{array}{c|cc|} & \tau_{d-} & \tau_{a-} \\ \tau_{d-} & 0 & -2 \\ \tau_{a-} & 5 & 0 \end{array}$$

(b)

$$\begin{array}{c|ccc|} & \tau_{d-} & \tau_{a-} & \tau_{b-} \\ \tau_{d-} & 0 & -2 & \infty \\ \tau_{a-} & 5 & 0 & -2 \\ \tau_{b-} & \infty & 5 & 0 \end{array}$$

(c)

$$\begin{array}{c|ccc|} & \tau_{d-} & \tau_{a-} & \tau_{b-} \\ \tau_{d-} & 0 & -2 & -4 \\ \tau_{a-} & 5 & 0 & -2 \\ \tau_{b-} & 10 & 5 & 0 \end{array}$$

(d)

$$\begin{array}{c|cccc|} & \tau_{d-} & \tau_{a-} & \tau_{b-} & \tau_{c-} \\ \tau_{d-} & 0 & -2 & -4 & \infty \\ \tau_{a-} & 5 & 0 & -2 & \infty \\ \tau_{b-} & 10 & 5 & 0 & -2 \\ \tau_{c-} & \infty & \infty & 5 & 0 \end{array}$$

(e)

$$\begin{array}{c|cccc|} & \tau_{d-} & \tau_{a-} & \tau_{b-} & \tau_{c-} \\ \tau_{d-} & 0 & -2 & -4 & -6 \\ \tau_{a-} & 5 & 0 & -2 & -4 \\ \tau_{b-} & 10 & 5 & 0 & -2 \\ \tau_{c-} & 15 & 10 & 5 & 0 \end{array}$$

(f)

Figure 3.6. Zone creation and evolution.

along all paths to that state has exceeded the maximum logic delay d. Therefore, if along a different path we encounter this state and did not stabilize, then this state transition cannot stabilize the node n. If the amount of accumulated delay does not exceed the delay d, then the algorithm must determine if it is going to recurse down this state transition. If this state has been seen previously upon this path, the algorithm has encountered a cycle of states and must not recurse. If the Boolean evaluation of the node n has changed, then again the algorithm must not recurse. If this is a new state on this path and the Boolean evaluation is maintained, then the algorithm recursively visits the state s'. Note that this edge may have been found to be stable along a different path, but it is not stable along the path the algorithm is currently working on. Therefore, the algorithm must say this edge is not stable before recursing. Although this algorithm can potentially explore a lot of paths, we have not seen this happen in our experimental results. Typically, the length of the paths explored is very short as the recursion is terminated quickly. If in the future, we find examples where this is not the case, we can limit the path length to improve efficiency at the potential cost of more false negative results.

Let us consider again the example netlist in Figure 2.3(b). A change in evalua-

```
do_timed(TPN,SG,NET,n,s,z,t_i,d,visit,path) {
  visit(s) = TRUE
  path(s) = TRUE
  foreach (s,t_j,s') in δ
    z' = update_zone(TPN,NET,z,t_j)
    if (-1*z'_ij > d) then
      if (not visit(s')) then
        stable(s,s',n) = TRUE
    else if (not path(s') &&
            eval(s,n) == eval(s',n)) then
      stable(s,s',n) = FALSE
      do_timed(TPN,SG,NET,n,s',z',t_i,d,visit,path)
  path(s) = FALSE
}
```

Figure 3.7. Timed stabilization recursion.

tion in node e occurs between states 1110 and 0110. As mentioned previously, the `do_timed` function is called with the zone shown in Figure 3.6(b). As we traverse the SG, the next transition encountered is $b-$. Since $b-$ fires 2 to 5 time units after $a-$, these entries are entered into the appropriate rows and columns as shown in Figure 3.6(c). The timing of the other non-diagonal entries are set to ∞. The zone is then recanonicalized using Floyd's all-pairs shortest-path algorithm, and the resulting zone is shown in Figure 3.6(d). We are interested in the minimum elapsed time between the last transition entered, $b-$, and the initial transition $a-$ which is 2 in this case. Note that lower bounds appear as negative values in a DBM. Since two time units is insufficient time to say with certainty that node e has stabilized, the algorithm considers recursing on state 0010. Since this state has not yet been explored on this path, and since node e still evaluates to 0 in this state, the algorithm recurses to state 0010. Upon recursion, the algorithm adds transition $c-$ to the zone as shown in Figure 3.6(e) and recanonicalizes to obtain the zone shown in Figure 3.6(f). The new minimum time elapsed from $a-$ till $c-$ is 4 time units. Since this number is larger than the maximum delay of the AND gate (2 time units), the algorithm can mark this edge as stabilized. The `propagate` function copies this stabilization onto states 0000, 0100, and 1000 and edges $(0000, b+/1, 0100)$,

$(0100, c+/1, 0110)$, $(0000, a+, 1000)$, and $(1000, b+/2, 1100)$. This is significant in that the hazard condition that existed after untimed stabilization cannot manifest because of the timing relationships between the circuit and the SG.

3.4 Checking for hazard-freedom

Hazards can manifest in asynchronous circuits due to violations in the *acknowledgment* or *monotonicity* properties [4]. This section gives an explanation of how our method checks for violations in acknowledgment and monotonicity. This explanation though, is brief, and high-level since it is essentially the same as that described in [4].

An acknowledgment violation occurs when an internal node becomes excited to change to a new value, but its excitation changes value before it can be shown to have stabilized. The algorithm shown in Figure 3.8 uses the stability information found earlier to check for acknowledgment on all excited nodes. The algorithm examines each node n and each state transition (s, t, s') in which n changes Boolean evaluation. If n is not stable before it changes Boolean evaluation, then an acknowledgment hazard is reported. For the netlist shown in Figure 2.3(b) using only untimed stabilization, a hazard is found on node e for the state transition $(1000, b+/2, 1100)$. However, since timed stabilization detects that this edge is stable for node e, this circuit is hazard-free when timing is considered.

```
check_acknowledgment(SG,NET) {
  foreach (n ∈ N)
    foreach (s,t,s') ∈ δ
      if ((eval(s,n) ≠ eval(s',n)) and
        not stable(s,t,s')) then
         report hazard on n for (s,t,s')
}
```

Figure 3.8. Algorithm to check acknowledgment.

A monotonicity violation occurs when an internal or output node is supposed to remain stable but it becomes momentarily excited or it is supposed to make a transition which it makes non-monotonically. This occurs when a gate has a *potential hazard* while there is no stable, *forcing* side-input. For example, a potential hazard exists when the output of an AND gate is supposed to remain stable low or fall, but one input is rising. If a side-input cannot be found that is stable low while the other input is rising, it is possible that the AND gate may momentarily evaluate to 1 causing a glitch on the output of the AND gate. The algorithm to check for monotonicity violations is given in Figure 3.9. The definitions of `potential_hazard` and `forcing` are a bit involved and can be found in [4].

```
check_monotonicity(SG,NET) {
  foreach (n ∈ N)
    foreach (s ∈ S)
      foreach v ∈ FI(n)
        if (potential_hazard(s,n,v)) then
           if (not exist w such that forcing(s,n,w)
              and stable(s,w)) then
              report hazard on n for (s,v)
}
```

Figure 3.9. Algorithm to check monotonicity.