Comparative Results for Verification								
			Our Tool			AVER		
Circuit	$\|N\|$	$\|V\|$	$\|\Phi\|$	Run-Time	Correct	$\|Q\|$	Run-Time	Correct
and-vs-c	0	1	8	0.1 sec.	NO	8	$<$ 1 sec.	NO
vbe10b	8	26	256	2.0 sec.	YES	3492	13 sec.	YES
pe-send-ifc	28	43	117	1.4 sec.	YES	43,296	8 min. 29 sec.	YES
wrdatab	12	28	216	1.7 sec.	YES	8976	56 sec.	YES
mp-forward-pkt	5	18	22	0.1 sec.	YES	70	$<$ 1 sec.	YES
pe-rcv-ifc	29	47	117	0.6 sec.	YES	27,131	4 min. 42 sec.	YES
vbe6a	8	24	192	1.1 sec.	YES	2448	9 sec.	YES
nak-pa	4	20	58	0.3 sec.	YES	123	$<$ 1 sec.	YES
ram-read-sbuf	5	22	39	0.2 sec.	YES	148	$<$ 1 sec.	YES
sbuf-read-ctl	2	15	19	0.1 sec.	YES	28	$<$ 1 sec.	YES
sbuf-send-pkt2	8	22	26	0.2 sec.	YES	332	2 sec.	YES
rcv-setup	1	8	11	0.1 sec.	YES	14	$<$ 1 sec.	YES
abcs	86	118	161	5.1 sec.	NO	???	???	???

Table 1: Comparative results.

can compare the number of states examined. In the largest example *pe-send-ifc*, AVER needs to examine 370 times as many states.

Moreover, the circuit *abcs*, an asynchronous controller designed in conjunction with Hewlett-Packard Labs, was simply too large for AVER to verify; AVER halted due to lack of memory after examining approximately 24,000 circuit states. Our program handled it quite easily detecting a complex-gate equivalence violation caused by a bug in the synthesis tool SYN [5]. With the violation description reported by our tool, SYN was corrected.

In addition, the circuit *and-vs-c* illustrates the significance of our progress condition based on complex-gate equivalence. The 8-state specification can be implemented correctly by a two-input AND gate. However, in this example we verify a C-element implementation instead. It is easy to see that the C-element satisfies the specification but does not exhibit all the behaviors in the specification. As a result, the C-element implementation is not complex-gate equivalent to the specification. It is interesting to note that even though AVER does not explicitly check complex-gate equivalence, AVER can still provide the same result. AVER reports that the circuit conforms to the specification as it does not require the circuit to have all behaviors. However, in addition we used AVER to check if the specification conforms to the implementation, in essence, checking the environment against the circuit. When we do this, AVER detects the problem. Similarly, our tool will find deadlock conditions that are not present in the specification which AVER will require two passes to detect.

In conclusion, the major contribution of the paper is a low complexity algorithm based on inferred values of internal signals for verifying determinate speed-independent circuits. The verification algorithm tests whether a circuit satisfies and is complex-gate equivalent to its specification. The algorithm is linear with respect to size of the SG specification while in comparison AVER and VERDECT uses algorithms which are linear with respect the size of the implementation SG, which can be exponentially larger. The algorithm can give false negatives but none were found, implying that our correctness conditions are sufficiently tight in practice. Our future is to expand the class of circuits we can verify.

Acknowledgements

We would like to acknowledge Chris Myers of Stanford University for numerous insightful discussions on this topic. We also would like to thank David Dill of Stanford University for his steady support and helpful advice.

References

[1] D. L. Dill. "Trace Theory for Automatic Hierarchial Verification of Speed-Independent Circuits". ACM Distinguished Dissertations, 1989.

[2] J. C. Ebergen. "A Verifier for Network Decompositions of Command-Based Sspecifictions". In *Proc. of the 26th Annual HICSS*, pages 310–318. IEEE Computer Society Press, 1993.

[3] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky. "Analysis and Identification of Speed-Independent Circuits on an Event Model". To be published in *Formal Methods in System Design*.

[4] Srinivas Devadas, Kurt Keutzer, Sharad Malik, and Albert Wang. "Verification of Asynchronous Interface Circuits with Bounded Wire Delays", October 1993. To Appear in *Journal of VLSI Signal Processing*.

[5] Peter A. Beerel and Teresa H. Y. Meng. "Automatic Gate-Level Sythesis of Speed-Independent Circuits". In *Proceedings IEEE 1992 ICCAD* Digest of Papers, pages 581–586, 1992.

[6] Jerry R. Burch. *Trace Algebra for Automatic Verification of Real-Time Concurrent Systems*. PhD thesis, Carnegie Mellon University, 1992.

[7] P. C. McGeer and Robert K. Brayton. *Integrating Functional and Temporal Domains in Logic Design*. Kluwer Acadamic Publishers, Norwell, MA, 1991.

```
            (stable(u, s') == UNKNOWN and
             not stable_recur(u, s') ))
             stable(u, s) = FALSE
             return FALSE
    stable(u, s) = TRUE
    update sensitized(w, v, s) for all (w, v) ∈ E, v ∈ FO(u)
    return TRUE
}
```

Notice that *must_fire*(u, s', s) is defined in terms of *sensitized*(u, v, s) for $v \in FO(u)$ which, in turn, is defined in terms of *stable*(w, s), where $w \in FI(v)$. Hence w and u are both fanins of v and each of their stability depends the stability of the other. If we are not careful as to the order that the stability of these fanin signals are checked, we might too conservatively estimate *stable*(w, s). This is why we execute *stable_recur*(u, s) for all the fanin signals twice. Because signal regions sometimes contain cycles of states we use an array *seen*(s) to keep track of those states we have already examined and thereby avoid an infinite loop. The proof that the results of this algorithm is the least fixed point of *stable* is relatively straightforward.

4.2 Complexity of *find_stable_states*(G, M)

This section provides a complexity analysis of our algorithm to compute *stable*(u, s). In this analysis we assume constant-time look-up for all tables.

The function *find_stable_states*(G, M) calls the function *stable_recur*(u, s) no more than $2|\Phi|$ times for each $u \in V$. Even though *stable_recur*(u, s) also calls itself recursively, because of the dynamic programming function style, it is called a total of $O(\Gamma)$ times for each $u \in V$. Therefore, the function *must_fire*(u, s', s) is called a total of $O(V \cdot \Gamma)$ times. Because we call *stable_recur*(u, s) for each circuit signal in reverse topological order, we can also use dynamic programming to calculate *must_fire*(u, s, s'). This allows us to calculate it by examining only the immediate fanouts of u. Hence, the function *must_fire*(u, s', s) contributes a total of $O(E \cdot \Gamma)$ time to *find_stable_states*(G, M).

The function *sensitized*(u, v, s) is implemented with a look up table as follows. According to the definition of *sensitized*(u, v, s) it is easy to see that the function depends only on *eval*(w, s) for all $w \in FI(v)$ and the stability of all side-inputs of u. The look up table is first initialized assuming external signals are always stable and internal signals are never stable. Each time an internal signal u is made stable in state s, the entries *sensitized*(w, v, s) for all $w \in FI(v)$ with $v \in FO(u)$ are updated. This has a total time contribution of $O(\Phi \cdot E \cdot p)$, where p is the maximum fanin of any signal of the circuit.

Summing the time contributions, we find the function *find_stable_states*(G, M) takes a total of $O(\Phi \cdot V + E \cdot \Gamma + \Phi \cdot E \cdot p)$ time. This is the key to the low-complexity of our verification algorithm. In affect, we are able to conservatively infer the stability of internal signals in SG states without ever examining implementation-states explicitly, thereby avoiding the state explosion problem caused by internal signals.

4.3 A verification algorithm

This subsection provides our algorithm to check the sufficient conditions developed.

Algorithm 4.3 *(Check circuit)*

```
check_correctness(G, M) {
    find_stable_states(G, M)
    foreach u ∈ O
        if (not check_equivalence(G, M, u))
            return FALSE
    foreach u ∈ N ∪ O
        if (not check_monotonicity(G, M, u) or
            not check_acknowledgement(G, M, u))
            return FALSE
    return TRUE
}
```

The function *find_stable_states*(G, M) creates a look-up table of stable signals and states which is used by the Boolean functions *check_monotonicity*(G, M, u) and *check_acknowledgement*(G, M, u). The implementation of *check_equivalence*(G, M, u) is straightforward from the definition of complex-gate equivalent. The implementation of *check_monotonicity*(G, M, u) is a straightforward application of the definition of monotonicity. Determining whether or not *forcing*(u, v, s) and *potential_hazard*(u, v, s) holds for basic gates such as ANDs and ORs is trivial depending only on the evaluation of u and v. Determining these functions for C-elements is only slightly more complicated. Because all memory elements are restricted to be attached to a primary output the current memory value is always known. Hence, in each state s a C-element acts like either an AND or an OR gate. The implementation of *check_acknowledgement*(G, M, u) is also rather straightforward except that all region paths of each region of u are checked simultaneously using a depth-first-search.

4.4 Complexity of *check_correctness*(G, M)

Using the complexity result of *find_stable_states*(G, M), it is easily verified that *check_correctness*(G, M) takes $O(\Phi \cdot V + E \cdot \Gamma + \Phi \cdot E \cdot p)$ time, which is linear with respect to the size of the SG specification. It is also linear with respect to the size implementation, assuming a constant maximum fanin of any circuit signal.

4.5 When false negatives are possible

Our verification algorithm is conservative because in some cases false negatives can arise. They arise for two reasons.

First, our notion of stability is conservative. In circuits with tight reconvergent fanout, our algorithm to find the stability may not find some signals to be stable when in fact they can not change. More exact algorithms exist but with the cost of added complexity.

Second, our sufficient conditions require that when a potential hazard exists that a forcing signal must also be stable. In some circuits a forcing signal need not be stable to prevent the hazard. As long as a forcing signal is ensured to fire before the hazard-causing signal fires, the hazard is avoided. This can occur when there is internal causality between the disabling signal and the forcing signal. Moreover, with some complex gates, such as threshold gates, no single signal can be forcing and hence the sufficient conditions for correctness will often be violated. For such gates, however, sets of signals can be forcing. In other words, if all signals in such a set were stable, the hazard is avoided. Extending the sufficient conditions to allow sets of forcing signals is not difficult, but adds significant complexity to the corresponding verification algorithm.

5 Experimental results and conclusions

Table 1 presents a comparison between an experimental version of our tool and AVER for a large benchmark of speed-independent circuits, many of which were created by the synthesis tool SYN [5] from a suite of industrial and academic specifications provided by Luciano Lavagno. The comparison was done using a DEC Station 5000. The columns $|N|, |V|$, and $|\Phi|$ give the number of internal signals, number of circuit signals, and the number of SG states. The column $|Q|$ is the number of implementation-states AVER examines.

In evaluating the comparison it is important to note that AVER is written in compiled LISP and our tool is written in C. Using a degrading factor of 20 for compiled Lucid Common LISP, our program is still between 18 and 23.5 times faster than AVER in the largest two examples *pe-send-ifc* and *pe-rcv-ifc*. To further demonstrate the complexity advantage of our tool we

Examples of potential hazards are when (a) an OR gate, enabled to rise, has an input which is falling, (b) an AND gate which has been enabled to fall has an input which is rising, (c) an OR gate, enabled to fall, has an input which is rising, and (d) an AND gate which is enabled to rise has a falling input.

The first two situations might or might not cause a hazard. This is because the signal transition that might be disabled is disjunctive; it's enabling can be maintained by other inputs. The last two always create a failure. This is because the signal transition that is disabled is conjunctive, depending on all inputs to maintain its enabling.

More formally we provide the following definitions.

Definition 3.5 (Disabling Signal) *The signal v is a disabling signal of u, $v \in \text{FI}(u)$, in state q, denoted disabling(v,u,q), when enabled(v,q), enabled(u,q), and not enabled$(u, bitcomp(v,q))$.*

Definition 3.6 (Enabling Signal) *The signal v is an enabling signal of u, $v \in \text{FI}(u)$, in state q, denoted enabling(v,u,q), when enabled(v,q), not enabled(u,q), and enabled$(u, bitcomp(v,q))$.*

Definition 3.7 (Potential hazard) *The signal u has a potential hazard caused by signal v in specification-state s, denoted potential_hazard(v,u,s), if there exists an implementation-state $q \in proj^{-1}(A')(s)$ such that (a) eval$(u, proj(A)(q)) =$ eval$(u, proj(A)(bitcomp(v,q)))$, (b) if $v \in N$, then at_eval(v,q) and not stable(v,s), and (c) either*

- *not at_eval(u,q) and disabling$(v,u,bitcomp(v,q))$ or*
- *at_eval(u,q) and enabling$(v,u,bitcomp(v,q))$.*

A *forcing input* for a rising (falling) transition of u is a signal $v \in FI(u)$ which when equal to its evaluation maintains the enabling of u. For example in a 2-input OR gate if u and v evaluate to 1, then v is said to be a forcing input of u. Notice that conjunctive transitions, such as the falling transition of a 2-input OR gate, do not have forcing inputs because the transition of v requires all fanins to be equal to their evaluation before it is enabled.

More formally we provide the following definition of a forcing signal.

Definition 3.8 (Forcing signal) *The signal v is a forcing signal of u, denoted forcing(v,u,s), if $v \in \text{FI}(u)$ and for all $q \in proj^{-1}(A')(s)$ we have that at_eval(v,q) implies (at_eval(u,q) iff not enabled(u,q)).*

Identifying a signal v as a forcing signal of u is important because when v is stable in some state s, it guarantees that u will not have a hazard in that state.

3.3 Guaranteeing correctness

Our sufficient conditions for correctness are a formalization of the following observation: a correct circuit is one in which for every SG region path the corresponding signal exhibits the corresponding monotonic transition (rising if $eval(u,s) = 1$ in all states s in the region, falling if $eval(u,s) = 0$ in all states s in the region).

We first define the notion of monotonicity of a region path which plays a key role in guaranteeing that u will be hazard-free in all feasible traces.

Definition 3.9 (Monotonic region path) *A region path $P = s_1 s_2 \ldots s_p$ of signal u is said to be monotonic if in every state $s_i, i \in \{1,\ldots,p\}$ in which there exists a potential hazard, potential_hazard(v,u,s_i), there also exists a forcing signal w of u in s_i which satisfies stable(w,s_i).*

Next, we define the notion of an acknowledged region path of a signal u which contains a SG state transition that guarantees that u will fire at least once in that path.

Definition 3.10 (Acked region path) *A region path $P = s_1 s_2 \ldots s_p$ of signal u is said to be acked if must_fire(u, s_i, s_{i+1}) holds for some transition $s_i \rightarrow s_{i+1}$, $i \in \{1,2,\ldots,p-1\}$.*

Finally, we can present the major result of this paper whose proof is omitted due to space constraints.

Theorem 2 (Sufficient conditions for correctness) *Let $G = \langle V, E\rangle$ be a circuit implementing state graph $M = \langle I, O, \Phi, \Gamma, s_o\rangle$. Let Π be the set of region paths for all signals $v \in N \cup O$. If G is complex-gate equivalent to M and all $P \in \Pi$ are acked and monotonic, then the circuit is correct.*

Hence, by checking simple properties of all region paths of the specification, circuit correctness can be inferred. In the next section, we will show how these properties depend on local properties of states and state transitions and hence can be efficiently verified.

4 Verification

This section provides a linear-time algorithm for verifying the correctness of a determinate speed-independent circuit.

4.1 An algorithm to determine stability

First, we provide an efficient algorithm to precompute *stable*(u,s) for all $u \in V$ and $s \in \Phi$. Consider the signal region of u which contains the state s. The definition of *stable*(u,s) states that if there exists a path from an entrance state to s in which no SG transition is one in u must fire, then u may not be stable. Otherwise u is stable.

Our proposed algorithm searches the signal region of u backwards from state s for such a path. If no path is found then u is stable. Otherwise u is declared unstable. Notice that in both of these cases, the stability of u in all states of each path searched must be determined first. In other words, the algorithm can use dynamic programming in which partial results are stored and used later. The details of the algorithm are given below.

Algorithm 4.1 *(Finding stable states)*

```
find_stable_states(G, M) {
    topologically sort V
    find evaluation for all v ∈ V and s ∈ Φ
    initialize stable(v, s) to UNKNOWN for all v ∈ N, s ∈ Φ
    initialize stable(v, s) to TRUE for all v ∈ I ∪ O, s ∈ Φ
    initialize sensitized(u, v, s) for all (u, v) ∈ E, s ∈ Φ
    foreach s ∈ Φ
        foreach u ∈ V in reverse topological order
            foreach v ∈ FI(u)
                initialize seen array
                stable(v, s) = stable_recur(v, s)
            foreach v ∈ FI(u)
                if ( stable(v, s) == FALSE )
                    initialize seen array
                    stable(v, s) = UNKNOWN
                    stable(v, s) = stable_recur(v, s)
}
```

Algorithm 4.2 *(Find stable states—recursive step)*

```
stable_recur(u, s) {
    foreach s' ∈ Φ such that s' → s and not seen(s')
        seen(s') = TRUE
        if ( not must_fire(u, s', s) )
            if (eval(s', u) ≠ eval(s, u))
                return FALSE
            else if (( stable(u, s') == FALSE ) or
```

G with respect to M can now be defined. G *satisfies* M iff *init*$(\{s_0\})$(proj$(A)(T')) \subseteq T$. G is *complex-gate equivalent* to M exactly when for all $u \in O$ and all $s \in S$ that *eval*$(u, s) \neq s(u)$ iff *enabled*(u, s). G is a *correct* implementation of M if G satisfies and is complex-gate equivalent to M.

Our definition of satisfies is similar to the safety property trace conformance introduced by Dill [1] for this variant of trace theory. Complex-gate equivalence, on the other hand, is a liveness property. It ensures that if all gates in a correct circuit are collapsed into one complex gate, thereby removing all internal signals, the resulting circuit must have the same behavior as its specification. This means that our correctness criterion detects circuit deadlock conditions not present in the specification.

3 Correctness conditions

First, we develop necessary and sufficient conditions which relate satisfies to our notion of hazard-freedom. The set *proj* $^{-1}(A')(S) \cap P'$ is the set of *feasible* traces, i.e., traces executed with the environment behaving according to the SG specification. We define Q to be the set of states reachable by feasible traces, $Q = \{q \mid xq \in$ *proj* $^{-1}(A')(S) \cap P'\}$. For a state $q \in Q$, we say $q \rightarrow q'$ if xqq' is a feasible trace. The notation $q \xrightarrow{v} q'$ may also be used if, in addition, $q' =$ *bitcomp*(v, q). We can now present the following result whose proof is omitted due to space constraints.

Theorem 1 *Let the circuit $G = \langle V, E\rangle$ be complex-gate equivalent to the state graph $M = \langle I, O, \Phi, \Gamma, s_\circ\rangle$. Let T be the trace structure for M and T' be the trace structure for G. G satisfies M iff for all states $q \in Q$ and all transitions $q \xrightarrow{v} q'$, for all $u \in N \cup O$ if enabled(u, q) then either enabled(u, q') or $u \neq v$.*

Intuitively, this theorem means that when a circuit is complex-gate equivalent to a state graph, checking whether the circuit satisfies the specification is equivalent to checking whether the circuit is hazard-free.

In the following subsections this result is used to develop sufficient conditions for correctness. Our approach involves characterizing circuit signals in SG states, even though some circuit signals (internal signals) are not specifically modeled in SG states. Since internal signals are not fixed in a SG state, a priori they can be 0, 1, or changing. However, we can sometimes infer that an internal signal is *stable* at either a logical 0 or 1 in a SG state, reducing the portion of the state space we need examine exponentially.

In addition, we can define relationships between pairs of signals in SG states such as notions of sensitized, forcing, and disabling. These notions will be used to identify all potential hazards and describe a means of guaranteeing that these hazards cannot occur.

3.1 Stable and Sensitized signals

In combinational circuits the notion of sensitized signals has been well studied [7]. The well-known conditions in which a change in signal u causes a change in signal v in state s is defined by the Boolean difference function $\frac{\partial f}{\partial u}(s)$. Boolean difference $\frac{\partial f}{\partial u}(s)$ tells us what values we must assert on all other fanins of v (also referred to as side-inputs in [7]) to make the value of v sensitized to the value of u in some state s. Hence, in our notation, if $f =$ AND(u, w), then $\frac{\partial f}{\partial u}(s) =$ *eval*(w, s). However this sensitization is guaranteed only after all signals have settled down.

In an asynchronous circuit, however, in addition to the settled down values, the transient behavior of the signals is relevant when considering hazards. Therefore, for the purposes of this paper, v is sensitized to u in state s when v is sensitized to u in all *possible* implementation-states that project onto s.

In determining which implementation-states are possible, we assume that side-inputs that are *stable* are fixed at their evaluated value and that all other inputs can be either 0 or 1. A signal is stable in state s if every state $q \in Q$ which projects onto s satisfies q(u) = *ext*(u)(q). By definition of *ext*(u), external signals are stable in all states.

However, checking the stability for internal signals involves finding all states in Q which is too computationally expensive. Instead, we use a least fixed point calculation which finds a subset of all pairs (u, s) of which u is stable in s given in the following four mutually recursive definitions.

Definition 3.1 (Stable set) *The set of pairs of signals and SG states in which they are stable is defined as follows.*

- *For all $s \in \Phi$ and $u \in I \cup O$, $(u, s) \in$ stable.*
- *If all region paths of $u \in N$ which include s (a) do not end in s and (b) first go through a state transition $s' \rightarrow s''$ for which must_fire(u, s', s''), then $(u, s) \in$ stable.*

Definition 3.2 (*must_fire*(u, s, s')) *A signal u is said to must fire between states s and s', denoted must_fire(u, s, s'), when $s \rightarrow s'$ and either*

- *$u \in N$ and sensitized(u, v, s) for some $v \in$ FO(u) for which must_fire(v, s, s'), or*
- *$u \in O$ and $s \xrightarrow{u} s'$*

holds.

Definition 3.3 (Strong inverse projection) *The strong inverse projection of a specification-state s, denoted proj$^{\dagger}(A')(s)$, is the set of implementation-states which project onto the specification-state s given the set of signals that are known to be stable in s. More formally,*

proj$^{\dagger}(A')(s) = \{q \in 2^{I \cup O \cup N} \mid$ proj$(A)(q) = s$ and for all $u \in N, ((u, s) \in$ stable$) \Rightarrow$ at_eval$(u, q)\}$.

Definition 3.4 (Sensitized signal) *The signal u is a sensitized input of v in state s when for all $q \in$ proj$^{\dagger}(A')(s)$ we have that if (at_eval(v, q) iff not enabled(v, q)) then at_eval(u, q).*

It can be shown that this definition is well-founded. When the circuit is determined to be correct, it can also be shown that the least fixed point defined by these definitions is a conservative approximation of stability. For convenience we often write *stable*(u, s) to denote $(u, s) \in$ *stable*.

The function *must_fire* has a nested induction on the level of the circuit. Intuitively, *must_fire*(u, s, s') states that given only the signals currently known to be stable, assuming nothing about all other signals, that signal u is ensured to fire before the state transition $s \rightarrow s'$ completes. It is inductively defined on the level of the circuit where the inductive case states that a signal is not ensured to fire unless it is a sensitized fanin of a signal that must fire. The induction is easily seen to be well-founded.

3.2 Transitions

This subsection characterizes all causes of hazards. If a change of signal u might cause a hazard on signal v in some SG state s by driving v from an enabled state to an unenabled state, then we say there is a potential hazard, denoted *potential_hazard*(u, v, s). By identifying all potential hazards in SG states rather than in implementation-states we avoid explicitly examining implementation-states and thereby obtain a low-complexity verification algorithm.

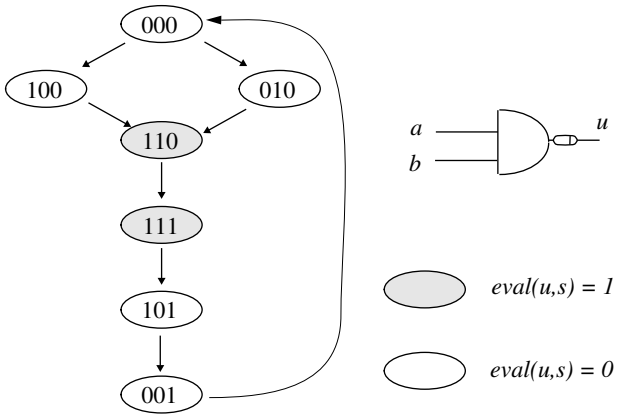

Figure 3: Example of evaluation and signal regions.

of SG states in which a signal evaluates to the same value. A semiconnected set of states R is one in which if $s, s' \in R$ then there exists either a SG state path $s \rightarrow s_1 \rightarrow \cdots \rightarrow s_n \rightarrow s'$ or a SG state path $s' \rightarrow s_1 \rightarrow \cdots \rightarrow s_n \rightarrow s$ such that $s_i \in R$ for $i = 1, \ldots, n$. For each signal u, the SG states are partitioned into a set of signal regions which correspond to different signal transitions. In Figure 3, one can see two signal regions of u, a rising region $R_1 = \{110, 111\}$ and a falling signal region $R_2 = \{000, 100, 010, 101, 001\}$. Paths through such regions are defined below.

Definition 2.1 (Region path) *Given a SG, a region path P for a signal region R of a signal u is a sequence of SG states $s_1 s_2 \cdots s_n$ such that $s_i \rightarrow s_{i+1}$ holds for $i = 1, \ldots, n-1$, $s_i \in R$ for $i = 1, \ldots n-1$, $s_n \notin R$, and s_1 can be entered from a state not in R.*

For example, in Figure 3, there are three regions paths for the signal u. One region path is through the rising region, $P_1 = 110\ 111\ 101$, and two region paths are through the falling region, $P_2 = 101\ 001\ 000\ 010\ 110$ and $P_3 = 101\ 001\ 000\ 100\ 110$.

2.4 Trace theory

In order to reason about correctness of circuits with respect to their specifications, the behavior of both circuits and specifications must first be precisely modeled in a compatible form. One desirable feature of this form is to maintain a clear separation between the desired and actual behavior of the circuit. To do this, we use a variant of the prefix-closed trace theory as described by Dill in [1].

In our model, traces are defined as sequences of states rather than transitions. The set of traces $\mathcal{B}(A)$ is generated over the alphabet of signals names $A = I \cup O$, and includes all sequences $x \in \left(2^A\right)^*$ in which any two adjacent states differ on exactly one signal. A trace structure is a 4-tuple $\langle I, O, S, F \rangle$. It defines the behavior of a system with inputs I and outputs O. The set of possible traces P is defined as the union of the set of success S and failure F traces, i.e., $P = S \cup F$.

The set P must be non-empty and receptive, which means that the system cannot prevent an input transition from occurring. That is, if $xs \in P$ then $xss' \in P$ for all s' such that $xss' \in \mathcal{B}(A)$ and there exists $a \in I$ for which $s(a) \neq s'(a)$. In addition S and P must be prefix-closed, that is, any prefix of a trace in S (P) is also in S (P).

Projection of traces is defined as the natural extension of projection of states to sequences of states except that stutters, where adjacent states of the projected sequence are equal, are removed.

Whenever $O \cap O' = \emptyset$, two trace structures T and T' can be composed defining the behavior of the composite system in which output signals in T may drive input signals in T' and vice versa. $T \| T' = \langle I'', O'', S'', F'' \rangle$, where $O'' = O \cup O'$, $I'' = I \cup I' - O''$, $A'' = O'' \cup I''$, $S'' = \{x \in \mathcal{B}(A'') \mid \text{proj}(A)(x) \in S \text{ and } \text{proj}(A')(x) \in S'\}$, and $F'' = \{x \in \mathcal{B}(A'') \mid (\text{proj}(A)(x) \in F \text{ and } \text{proj}(A')(x) \in P') \text{ or } (\text{proj}(A)(x) \in P \text{ and } \text{proj}(A')(x) \in F')\}$.

Trace containment corresponds to the notion of language containment. Whenever, $I = I'$ and $O = O'$, T is contained in T', denoted $T \subseteq T'$, iff $F \subseteq F'$ and $P \subseteq P'$. $F \subseteq F'$ ensures that the new system has less failures than the old. $P \subseteq P'$ ensures that the new system cannot cause new failures when placed in an environment.

Finally, we define trace structure initialization which generates the behavior of a system that is constrained to start up in one of a set of known states Q_0. Whenever $Q_0 \subseteq 2^A$, $\mathit{init}(Q_0)(T) = \langle I, O, S'', F'' \rangle$, where $S'' = \{x \in S \mid x(0) \in Q_0\}$ and $F'' = \{x \in F \mid x(0) \in Q_0\}$.

2.5 Trace structure for a specification, gate, and circuit

Given a SG specification M, it is straightforward to define the set traces S that are generated by the runs of M. The set F, needed to complete the trace structure T for M, can be constructed from I, O, and S as follows. If $S = \emptyset$, then F is the set of all traces over alphabet A. Otherwise, F contains all traces y which have prefixes xs and xss' such that xs is a successful trace but $s \xrightarrow{v} s'$, $v \in I$, is a state transition not specified in M. In other words, traces in the failure set F are caused by illegal input signal transitions. Similarly, illegal output signal transitions cause traces that are not possible.

The trace structure for a gate g is the four tuple, $\langle I_g, O_g, S_g, F_g \rangle$. $I_g = FI(u) = \{v_1, \ldots v_n\}$ and $O_g = \{u\}$. The successful trace set S_g is defined as follows.

$$
\begin{aligned}
&\langle q \rangle \in S_g \Leftrightarrow f(q(u), q(v_1), \ldots, q(v_n) = q(u) \\
&xqq' \in S \Leftrightarrow \\
&\quad xq \in S_g \text{ and } xqq' \in \mathcal{B}(A) \text{ and} \\
&\quad \big([q(u) \neq q'(u) \text{ and } \mathit{enabled}(u, q)] \text{ or} \\
&\quad \big[q(u) = q'(u) \text{ and } (\mathit{enabled}(u, q) \Rightarrow \mathit{enabled}(u, q'))\big]\big),
\end{aligned}
$$

where $\mathit{enabled}(u, q)$ is the natural extension of the function *enabled* to states $q \in 2^{I_g \cup O_g}$. The failure trace set F_g is defined from I_g, O_g, and S_g similarly to the failure set of a specification. Intuitively, a failure trace is caused by an input signal transition driving the gate from an enabled state to an unenabled state, causing a *hazard*. In practice, a hazard can cause a voltage glitch at the output of the gate. Notice that this definition requires that node of the gates are enabled in the reset state of the circuit.

The behavior of the circuit is an open system defined by the composition of the trace structures of all gates in the circuit. Since we assume that the input and output signals of the circuit match that of the specification, the resulting trace structure is of the form $\langle I, O \cup N, S', F' \rangle$. The behavior of the circuit is thus defined independent of the specification.

2.6 Definition of correctness

Let $M = \langle I, O, \Phi, \Gamma, s_0 \rangle$ be a state graph specification and let $G = \langle V, E \rangle$ be a circuit. Let $T = \langle I, O, S, F \rangle$ be the trace structure for M and let $T' = \langle I, O \cup N, S', F' \rangle$ be the trace structure for the circuit G. Our notion of correctness of

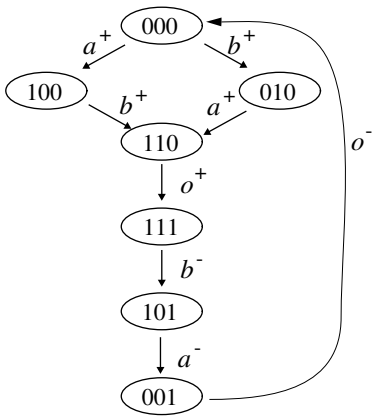

Figure 1: An example of a determinate speed-independen SG specification.

the implementation, consisting of input, output, and internal signals (N). Unless otherwise noted, from now on s will denote specification-states and q will denote implementation-states.

2.2 Specification

We specify a circuit with a low-level state graph (SG), M, defined with a tuple $\langle I, O, \Phi, \Gamma, s_o \rangle$, where Φ is a set of states, Γ is a set of edges, s_o is the start, or *reset*, state. An edge $e \in \Gamma$ is an ordered pair of states (s, s') which must differ in exactly one signal, i.e., $s' = \mathit{bitcomp}(s, u)$ for some $u \in I \cup O$. The signal s leads to s' (denoted $s \rightarrow s'$) if $(s, s') \in \Gamma$. The notation $s \xrightarrow{u} s'$ may be used when $s \rightarrow s'$ and $s' = \mathit{bitcomp}(s, u)$. We refer to $s \xrightarrow{u} s'$ as a state transition.

We say *enabled*(v, s) for $v \in I \cup O$ if v can change in state s, that is, if there exists an $s' \in \Phi$ such that $s \xrightarrow{v} s'$ holds. A signal which is enabled in s and not in s' for some $s \xrightarrow{v} s'$ is *disabled* by v. We require the SG specification to be determinate speed-independent, which means that an input signal can disable other input signals (input choice) but output signals can not disable any signals (no output choice). We refer the reader to [1, 2, 6] for works in which such output choice is allowed. A more formal definition is given below.

Definition 2.1 (Determinate speed-independent SG specification) *A SG $M = \langle I, O, \Phi, \Gamma, s_o \rangle$ represents a determinate speed-independent specification when for all $s, s' \in \Phi$ such that $s \xrightarrow{u} s'$, if $u \in O$ then*

- *for all $v \neq u$, if enabled(v, s) then enabled(v, s'),*

and if $u \in I$ then

- *for all $v \in O$, if enabled(v, s) then enabled(v, s').*

Figure 1 depicts an example of such a speed-independent SG specification. The SG is described by $M = \langle I, O, \Phi, \Gamma, s_o \rangle$, where $I = \{a, b\}$, $O = \{o\}$, $\Phi = \{000, 100, 010, 110, 111, 101, 001\}$, $\Gamma = \{(000, 100), (000, 010), (100, 110), (010, 110), (110, 111), (111, 101), (101, 001), (001, 000)\}$, and $s_0 = 000$.

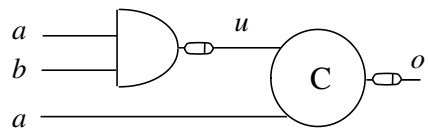

Figure 2: A circuit that implements the SG shown in Figure 1.

2.3 Implementation

In this paper, the output of all memory elements are required to be primary outputs and the primary outputs must cut the circuit, a criteria satisfied in the speed-independent implementations generated by an existing CAD tool [5]. A circuit G is defined with a directed graph $\langle V, E \rangle$. Each vertex $u \in V$ represents a circuit signal and each edge $e \in E$ represents a connection between signals. An edge e is directed, connecting a source vertex to a sink vertex. The set of fanins of u, denoted *FI*(u), is all vertices that are sources of edges that have u as its sink. Similarly the set of fanouts of u, denoted *FO*(u), is all vertices that are sinks of edges that have u as its source. Each gate output has an associated Boolean function $f(q(u), q(v_1), \ldots, q(v_r))$, where $\{v_1, \ldots, v_r\} = \mathit{FI}(u)$ and q is an implementation-state.

The vertices V can be partitioned into three pairwise disjoint sets I, N, O, where $V = I \cup N \cup O$, I is the set of the primary inputs, N is the set of internal signals, and O is the set of primary output signals. The *level* of $u \in V$ is defined recursively. If $u \in I \cup O$ the level of u is 0. Otherwise, the level of u equals one plus the maximum level of any of u's fanins. Because the circuit is cut by the primary outputs, this definition is well-founded.

Figure 2 depicts a circuit G that implements the SG shown in Figure 1. Formally $G = \langle V, E \rangle$, $V = I \cup N \cup O$ where $I = \{a, b\}$, $N = \{u\}$, and $O = \{o\}$, and $E = \{(a, u), (b, u), (u, o), (a, o)\}$. The functions associated with each gate output are f_o = C-ELEMENT(o, u, a) and f_u = AND(a, b).

We extend the definition of enabled of a signal u to implementation-states $q \in 2^{I \cup O \cup N}$. For $u \in N \cup O$, we say *enabled*(u, q) if u does not equal the value of the Boolean function of that signal in state q, i.e., $f(q(u), q(v_1), \ldots, q(v_m)) \neq q(u)$. For $u \in I$, we say *enabled*(u, q) iff *enabled*$(u, \mathrm{proj}(I \cup O)(q))$. That is, inputs are enabled in implementation-states iff they are enabled in the corresponding specification-states.

Each specification-state s has a unique implementation-state extension, *ext*(s), which projects onto s and in which every internal signals is not enabled. Intuitively, *ext*(s) is the implementation-state reached when the external signals are fixed and the internal signals are allowed to settle. The uniqueness of *ext*(s) follows from the facts that the primary outputs cut the circuit and that memory-elements are restricted to primary outputs.

For each specification-state s the value that signals tend to become in state s is denoted *eval*(u, s). For signals $u \in I \cup N$, *eval*$(u, s) =$ *ext*$(s)(u)$, i.e., the value that the input and internal signal tend equals the value to which they settle when external signals are fixed. For signals $u \in O$, *eval*$(u, s) =$ *ext*$(s)u$ iff not *enabled*$(u, \mathit{ext}(s))$, i.e., output signals tend to their settled values iff in the settled state *ext*(s) they are not enabled either. For brevity we define *at_eval*(u, q) as follows. *at_eval*(u, q) iff $q(u) =$ *eval*$(u, \mathrm{proj}(A)(q))$. Figure 3 depicts the evaluation of the internal signal u shown in Figure 2 in all SG states.

A signal region is defined as a maximal semiconnected set

Appeared in IEEE ICCAD-93 Digest of Technical Papers

Efficient Verification of Determinate Speed-Independent Circuits *

Peter A. Beerel
Computer Systems Laboratory
Stanford University
Stanford, CA 94305

Jerry Burch
Computer Science Department
Stanford University
Stanford, CA 94305

Teresa H.-Y. Meng
Computer Systems Laboratory
Stanford University
Stanford, CA 94305

Abstract

We present sufficient conditions for the correctness of speed-independent circuits with respect to their state graph (SG) specification, which can be tested in linear-time with respect to the size of the SG. Our correctness conditions consist of one safety condition and one progress condition. The progress condition detects deadlock conditions that are not present in the specification. The SG specifications considered are *determinate*, allowing input choice (conditionals) but not output choice (arbitration). The circuits considered are a network of basic gates; arbiters and mutual-exclusion elements are not allowed. We present an efficient algorithm to test the correctness conditions, in which false positives are not possible, but false negatives are possible. We have implemented the algorithm and present a table of run-time comparisons between our verification tool and the tool AVER by David Dill on a large benchmark of asynchronous circuits. The results demonstrate run-times of over an order of magnitude faster than AVER and no false negatives were found. Our speedup is achieved by avoiding the state explosion problem caused by explicitly examining the behavior of internal signals.

1 Introduction

The automated verification of speed-independent circuits has enjoyed significant success. Most notable is the tool AVER by Dill [1] which has detected errors in both unpublished and published designs. AVER uses trace theory to model the possible behaviors of the circuit, in which sets of traces are developed using finite automata to model the behavior of all signals in the circuit. However, the automata for highly concurrent circuits can be huge which degrades the efficiency of the verification, if not making it impossible. For even moderate size circuits (~100 gates), AVER can run out of space and quit.

There has been some successful work on reducing the verification complexity. Ebergen et. al. [2] (in the tool VERDECT) and Dill [1] propose, when possible, using stepwise or hierarchical verification to improve run-time efficiency. More recently, TRANAL, by Kishinevsky et. al. [3], uses an event-based specification of the circuit to avoid state explosion. This method dramatically reduces run-time for a speed-independence checker but does not check the circuit against a specification. It is also limited to circuits that have no input or output choice.

In addition, recent work [4] proposes verification of speed-independent circuits using run-based delay simulation and flow-graph equivalence checking. Although an exact complexity analysis is not given, the run-based simulation is known to be computationally expensive [4]. Moreover, this work uses a correlated delay model in which gate delays are fixed throughout the simulation, yielding the potential of false positives with respect to the speed-independent model.

*This research was supported in part by ONR Grant no. N00014-89-J-3036, by the Center for Integrated Systems, Stanford University, and by the Semiconductor Research Corporation, Contract no. 92-DJ-205.

In our method, a state graph (SG) specification is used to define the desired behavior of the primary inputs and outputs of a circuit. A circuit is defined to be a network of basic gates. Our verification algorithm tests one safety property (trace containment) and one progress property (complex-gate equivalence with the specification). The former property is similar to trace conformance defined by Dill [1]. The later property ensures that no deadlock would occur if no such behavior is specified in the circuit's specification SG. The verification is conservative; no false positives can occur, but false negatives are possible.

The complexity of the verification algorithm is linear with respect to the size of the specification SG (states plus transitions) and hence can be exponential in the number of specified signals. However, the size of the specification SG is often orders of magnitudes smaller than the implementation SG that needs to be searched by VERDECT and AVER. This is because implementation-states describe the value of all signals in the circuit, input, output *and* internal signals. Each internal signal in the circuit can potentially double the number of implementation-states that need to be analyzed. Hence, the implementation SG can be exponentially larger than the specification SG.

In this paper, our approach to reducing the verification complexity involves inferring the value of internal signals from those of external signals and using these inferred values to prove correctness. The cost of this complexity reduction is that the verification is conservative, not exact. We have run our tool on a large benchmark of circuits [5]. The run-time comparison demonstrates that our tool can be over an order of magnitude faster than AVER and no false negatives were found, which implies that our correctness conditions are sufficiently tight for realistic examples.

The organization of the paper is as follows. Section 2 provides notation and terminology used in this paper. Section 3 develops sufficient conditions for correctness. Section 4 presents an efficient algorithm of linear complexity with respect to the size of the specification SG to test these conditions. Section 5 gives a table of comparative run-times and our conclusions.

2 Notation and terminology

This paper deals with two levels of abstraction; that of a SG specification and that of a circuit. This section introduces a variety of definitions and concepts relating to these two levels of abstraction.

2.1 States

A state is a vector of binary values of some set of signals A, i.e., $s \in \{0,1\}^A$, or more succinctly, $s \in 2^A$. The value of a signal $u \in A$ in a state s is denoted by $s(u)$. The function *bitcomp*(s,u) returns the bitvector formed from s by complementing the bit corresponding to u. The projection of states between two sets of signals is defined as follows. For a state $q \in 2^A$ and $B \subseteq A$, proj$(B)(q)$ is another state s in which for all $v \in B, s(v) = q(v)$.

In this paper there will be primarily two sets of signals of interest. Those specified by the specification, consisting of input signals (I) and output signals (O), and those specified by