requests of each region-networks are *cut* by a falling transition of u. That is, every path through a falling request of a region-network contains a transition of u. Similarly, every path through a falling request of the OR gate in the set network is cut by a falling transition of u. It is easy to see that the falling transition of all region-networks and the OR gate in the set network are always required for the C-element to fall. Therefore u acknowledges their falling transition. Therefore, in every state transition between falling and rising requests u is stable at its external evaluation (by Lemma 2.1. In other words, the request for u is guaranteed to be acknowledged. The proof for the components in the reset network is analogous.

Second, we prove that the rising requests of all region networks and OR gates in set network S_u are acknowledged in the following steps:

1. Since the specification is determinate speed-independent, we know that every excitation region is cut by a transition of u. In fact, a transition of u is always the last transition in such a path.

2. We prove that the entrance constraint guarantees that every path through the rising request of a region-network includes a rising transition of u in two steps. First, recall that the entrance constraint guarantees that the cover of a region network A_i can only be entered through a state in the excitation region. Second, recall that every path from an excitation region state through the request must include a rising transition of u.

3. Next, we show that the combination of the entrance constraint and closure constraint ensures that the on-set (covers) of all region networks gates are disjoint. This means that if A_i externally evaluates to one in any state s that all other region networks A_j in the set network externally evaluate to 0 in this state.

4. This implies that for all other region networks A_j, s is part of falling request. Moreover, we can conclude that in every path through this falling request to state s there exists a falling transition of u.

5. We have already shown that the falling transition of u acknowledges the falling transition of all region-networks, which guarantees that all other region networks are stable 0 in state s (By Lemma 2.1).

6. Therefore the rising region-network is a required input to the rising OR gate in the set network and both the rising region-network and the rising OR gate are acknowledged by the rising transition of u.

The proof for the components in the reset network is analogous.

Now, we consider the monotonicity requirement. Monotonicity of the region networks is trivially satisfied because all inputs to the network are external. The monotonicity of the OR gates follows since when an OR gate input is unstable it is encouraging. The monotonicity of the C-element follows since the C-element is insensitive to any non-encouraging input because the other input is stable at a controlling value.

Since all gates satisfy monotonicity and acknowledgement, the circuit is correct. ∎

This theorem states the block-level implementation is correct if the covering constraints on all region functions are satisfied. The proof that the two presented binate covering algorithms correctly find correct covers because they are straight-forward and are thus omitted.

Our final proof responsibility is to prove that the logic transformations and optimizations preserve correctness. This is relatively easy since all non-trivial logic transformations (including the break-and-repair decomposition steps and logic optimizations) are accepted only if they pass the incremental verification algorithm. Hence, our proof responsibility reduces to showing that the incremental verification procedure is correct, which easily follows from the proof that the complete verification algorithm is correct [3].

3. Apply state and transition productions rules until a fixpoint is reached.

The resulting cube approximation corresponds to a Q_{CA} which contains q_0 and is a fixpoint. Hence, by Theorem 1 we have that $Q_{CA} \supseteq Q_{CEA}$. This algorithm uniquely defines the cube approximation fixpoint which is used to show that the standard C-implementation yields correct circuits.

E.2 Acknowledgement and stability

In order to prove correctness of our standard C-implementation, one of our proof obligations is to show that every request of every internal signal in the standard C-implementation is acknowledged. This is defined to mean that the cube approximation fixpoint must indicate that the internal signal u is stable in all state transitions between states of different requests of u. Our proof strategy is to use induction on the fixpoint definition. First, we show that acknowledgement holds in the initial cube approximation. Then, we show that no application of either production rule can ever invalidate acknowledgement.

To make this statement more precise we need to introduce the notion of an intermediate cube approximation. Let CA_i represent the cube approximation after i production rules have been applied after first being initialized according to steps 1 and 2 of the above algorithmic definition. Also, let CA represent the cube approximation fixpoint.

Lemma 2.1 *Let R be a signal region of internal signal u. If all state paths through R to state s'' include a transition $s \xrightarrow{v} s'$ for which acknowledges(s, v, u) holds in CA_i for all i then $CA_j(s'')(u) =$ ext_eval$(s)(u)$ for all j.*

Sketch of proof:
We prove this by induction on the length of the longest path from s' to s''.
Base case: (length of path is zero).
This is the case when $s' = s''$. For this case, $CA(s, v)(u)$ can never be set to X since *acknowledges*(s, v, u) guarantees that the guard for the transition production which can set $CA(s, v)(u)$ to x is always false. Since $CA_0(s, v)(u) =$ *ext_eval*$(s)(u)$, we conclude that $CA_j(s, v)(u) =$ *ext_eval*$(s)(u)$ for all j. As a result, the state production that sets $CA_i(s''')(u)$ to X can never fire. Since $CA_0(s'')(u) =$ *ext_eval*$(s)(u)$ we can conclude that $CA_j(s'')(u) =$ *ext_eval*$(s)(u)$ for all j.

Inductive case: Consider all s''' such that $s''' \xrightarrow{v'} s''$. Either $CA(s''')(u) =$ *ext_eval*$(s)(u)$ or *acknowledges*(s''', v', u) where $s''' \xrightarrow{v'} s''$. In both cases, the guard for the transition production that sets $CA(s''', v')$ is always false. Since $CA_0(s''', v')(u) =$ *ext_eval*$(s)(u)$, we conclude that $CA_j(s''', v')(u) =$ *ext_eval*$(s)(u)$ for all j. As a result, the state production that sets $CA_i(s''')(u)$ to X can never be applied. Since $CA_0(s'')(u) =$ *ext_eval*$(s)(u)$, we conclude that $CA_j(s'')(u) =$ *ext_eval*$(s)(u)$ for all j. ∎

E.3 Final proof of our synthesis algorithm

The final proof has two parts. First, we prove that our our standard C-implementations are correct block-level circuits. Second, we proof that the decomposition and optimization steps preserve correctness.

The critical step is to prove that the correct region function covers yield correct standard C-implementations.

Theorem 3 *If for all primary outputs $u \in O$ all region function covers $C(u, k)$ are correct and are implemented atomically, then the Standard C-implementation depicted in Figure 5 is correct.*

Proof: According to our definition of correctness we must show that the circuit implementation satisfies complex-gate equivalence and hazard-freedom.

Using only the cover condition of correct covers, simple Boolean logic verifies that the circuit is complex-gate equivalence.

Our proof strategy for proving that the circuit is hazard-free is to prove that the circuit satisfies the sufficient conditions for correctness described in Theorem 2. We show that there exists an output signal transition that acknowledges all requests and hence that every request is acknowledged. Moreover we show that all requests on all signals are monotonic.

First, we prove that the falling requests of all region networks and the single OR gate in the set network S_u are acknowledged by any falling transition of u. In a determinate speed-independent SG, every path between two rising transitions of u includes a falling transition of u. Therefore, the covering condition guarantees that, in every path between two region-network covers, a falling transition of u exists. Using this, it follows that the falling

w instead of the state cube *CA*(s). This provides a tighter condition since the transition $s \xrightarrow{v} s'$ could acknowledge some internal signals making them stable in *CA*(s,v). In addition, only the effect of v on u need be analyzed, since this is the only signal that changes in this specification state transition. Hence, sufficient conditions for monotonicity in specification transitions can be given as follows:

$$\textit{t-monotonic}(u,s,s') \quad \equiv \quad \textit{encouraging}(v,u,s) \ \vee \ \textit{insensitized}(v,u,s,\textit{CA}(s,v)) \vee v \notin \left(\textit{FI}(u) \cup \{u\}\right).$$

Guaranteeing that the signal is acknowledged before a particular specification state transition fires is easy since once any specification state transition acknowledges a signal, the signal remains stable until its external evaluation is changed. Specifically, an internal signal u is acknowledged before the state transition $s \xrightarrow{v} s'$ fires if u is stable in the transition cube *CA*(s,v).

$$\textit{acknowledged}(u,s,s') \quad \equiv \quad \textit{stable}(u,\textit{CA}(s,v)),$$

where $s \xrightarrow{v} s'$.

Together, monotonicity and acknowledgement provide the following sufficient conditions for correctness.

Theorem 2 (Sufficient conditions for correctness)

Let $G = \langle I, O, N, E\rangle$ be an externally-cut circuit implementing SG $M = \langle I, O, \Phi, \Gamma, s_0, \lambda\rangle$. If $Q_{CA} \supseteq Q_{CEA}$, G is complex-gate equivalent to M, and for all $s \in \Phi$ and all $u \in N \cup O$ s-monotonic(u,s) and for all $(s,s') \in \Gamma$ and all $u \in N \cup O$

$$\begin{aligned} \left[\textit{ext_eval}(u)(s) = \textit{ext_eval}(u)(s')\right] \quad &\Rightarrow \quad \textit{t-monotonic}(u,s,s') \\ \left[\textit{ext_eval}(u)(s) \neq \textit{ext_eval}(u)(s')\right] \quad &\Rightarrow \quad \textit{acknowledged}(u,s,s'), \end{aligned}$$

then the circuit is correct.

Proof: See [3]. ∎

E Proof of correctness

The design of speed-independent circuits has proven to be a tricky art. A published speed-independent circuit was later found to be hazardous [18] and we have shown that a published competing algorithm for the automatic design of speed-independent circuits [25] can be interpreted to allow for the synthesis of hazardous circuits. Both of these events motivate the need for a provably-correct synthesis algorithm. This section presents a proof that our standard C-implementation generates only correct circuits. Our proof strategy is to show that for our standard C-implementation their exists a cube approximation fixpoint for which the conditions in Theorem 2 are satisfied.

E.1 Defining a cube approximation fixpoint

Our sufficient conditions of monotonicity and acknowledgement (Theorem 2) can be applied to any cube approximation CA that corresponds to a set Q_{CA} that is a superset of Q_{CEA}. For a given circuit C there are many such cube approximations, some which may satisfy our tests of monotonicity and acknowledgement and some which are guaranteed to fail this test. For example, the cube approximation in which all internal signals are always unstable clearly represents a superset of Q_{CEA}, however, the acknowledgement test is guaranteed to fail. The problem is that this cube approximation is very poor, representing many unreachable states. To better check monotonicity and acknowledgement it is important to identify a good cube approximation (a close approximation of Q_{CEA}) on which the conditions of acknowledgement and monotonicity are tested.

Theorem 1 states that if the cube approximation is a fixpoint with respect to the given state and transition production rule and included q_0, then the cube approximation is guaranteed to be a superset of Q_{CEA}. However, the quality of the resulting fixpoint depends on the initialization of the cube approximation. It turns out that a good cube approximation can be obtained if each element in the cube approximation is initialized to its external evaluation.

More precisely, a cube approximation can be obtained from the following algorithm.

1. For all $s \in \Phi$ and for all $u \in N$ set *CA*(s)(u) = *ext_eval*$(s)(u)$.
2. For all state transitions $s \xrightarrow{v} s'$ in Γ and for all $u \in N$ set *CA*(s,v)(u) = *ext_eval*$(s)(u)$.

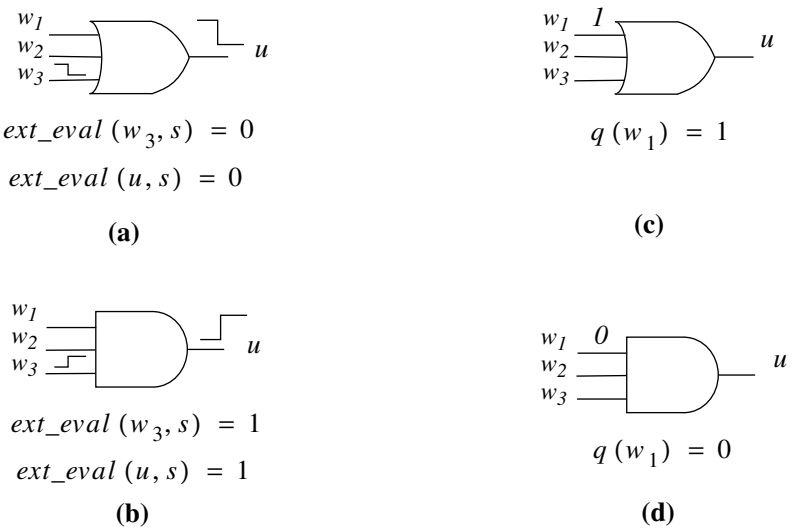


Figure 13: (a) and (b) Depiction of typical cases in which w_3 is an encouraging fanin of u. (c) and (d) Depiction of typical cases in which w_3 is an insensitized fanin of u.

The definition of encouraging fanins depends solely on the gate type and external evaluation of the relevant signals and can be easily hard-coded for simple gates.

The second condition is when the signal u is *insensitized* to the fanin w in a particular specification state or transition, i.e., when the behavior of f_u is independent of the actions of w given the current approximation of internal signals in the specification state or transition. Let the approximation of internal signals be represented by the cube c. Then, we formalize the notion of insensitized in a state s given the cube c based on the straight-forward notion of insensitized in implementation states as follows:

$$\begin{aligned} \textit{insensitized}(w,u,q) &\equiv w \in \textit{FI}(u) \ \wedge \ f_u(q) = f_u(\textit{bitcomp}(w,q)) \\ \textit{insensitized}(w,u,s,c) &\equiv \forall q \in \textit{CtoQ}(c,s) \ \left[\textit{insensitized}(w,u,q)\right] . \end{aligned}$$

As Figure 13(c) illustrates, w_3 is an insensitized fanin of u since the side-input w_1 is 1, a controlling value for the OR gate. Given a cube approximation, it is easy to see how this can be generalized to all simple gates and is very easy to check, taking $O(\textit{FI}(u))$ time to compute.

The third condition that ensures a fanin w does not violate the monotonicity of the signal u in a specification state s is when the fanin of u is stable in s. While cubes indicate when internal signals are stable in specification states, external signals are always stable in a specification state s, since when they change a new specification state is always entered. To express this, given a state or transition cube c, we say a signal is *stable* if it is fixed according to the cube or if it is an external signal, i.e.,

$$\textit{stable}(u,c) \equiv u \in I \cup O \ \vee \ c(u) \neq X.$$

Since a stable signal cannot change in any relevant implementation states that project onto s, it cannot cause u to violate monotonicity in this specification state.

We summarize these conditions in the following definition of monotonicity in specification states.

$$\begin{aligned} &\textit{s-monotonic}(u,s) \ \equiv \\ &\quad \forall w \in \left(\textit{FI}(u) \cup \{u\}\right) \left[\textit{stable}(w,s) \ \vee \textit{encouraging}(w,u,s) \ \vee \ \textit{insensitized}(w,u,s,\textit{CA}(s))\right] . \end{aligned}$$

Monotonicity must also hold during the transition between two specification states $s \xrightarrow{v} s'$ in which u's external evaluation does not change. In this case, the transition cube $\textit{CA}(s,v)$ can be used to check the insensitivity of a fanin

in s (assuming complex-gate equivalence). Second, it holds when the u is an internal signal which is stable in s. Summarizing, we have

$$\begin{aligned}&\textit{acknowledges}(s,v,w) \equiv \\ &\quad \exists u \in \textit{FO}(w)\left[\textit{required}(w,u,s,\textit{CA}(s)) \ \wedge \ \left[u = v \ \vee \ \textit{CA}(s,v)(u) = \textit{ext_eval}(s)(u)\right]\right].\end{aligned}$$

Using the notion of acknowledgement, we characterize a fixpoint of the the cube approximation $Q_{\textit{CA}}$ as one in which the following two production rules cannot be applied:

- **State production:** If $s \xrightarrow{v} s'$ and $\textit{CA}(s,v)(u) \neq \textit{CA}(s')(u)$, then set $\textit{CA}(s')(u)$ to X.
- **Transition production:** If $s \xrightarrow{v} s'$, $\textit{CA}(s)(u) = X$, and not *acknowledges*(s,v,u), then set $\textit{CA}(s,v)(u)$ to X.

State productions assert that if in a transition into some state s' an internal signal u is unstable, then u must be unstable in s'. State productions reflect the fact an internal signal is stable in a state only when it is stable during every transition into the state. Transition productions assert that if an internal signal is unstable in a state cube *and* the state transition does not *acknowledge* the internal signal, then the internal signal might still be changing during the transition. Transition productions reflect the fact that an internal signal becomes stable in a transition cube only when the transition acknowledges the internal signal. Notice that after the signal is acknowledged, however, it remains stable until its external evaluation changes.

Since the productions are monotonic and the approximation has a lower bound in which all internal signals are always X, a least fixpoint exists. Moreover, given an initial cube approximation, the least fixpoint is unique. We refer to this fixpoint as *CA* and to the corresponding implementation state set as $Q_{\textit{CA}}$.

We have proven in [3] that if a cube approximation implicitly contains the implementation state *extend*(s) for each specification state $s \in \Phi$ and is a fixpoint with respect to the above production rules that it represents a superset of the core externally-aligned states. This fact is made more explicit in the following theorem.

Theorem 1 (Superset theorem)
Let the circuit $G = \langle I, O, N, E\rangle$ be externally-cut and complex-gate equivalent to the SG $M = \langle I, O, \Phi, \Gamma, s_0, \lambda\rangle$. Let $Q_{\textit{CEA}}$ be the core set of externally-aligned states of G and M. Let $Q_{\textit{CA}}$ be the set of states represented by the cube approximation CA. If extend$(s) \in Q_{\textit{CA}}$ for all $s \in \Phi$ and no state or transition production rule can be applied to CA, then $Q_{\textit{CA}} \supseteq Q_{\textit{CEA}}$.

Proof: See [3]. ∎

D Sufficient conditions for correctness

This appendix develops sufficient conditions for correctness that are used to prove the correctness of our synthesis algorithm. These conditions are derived by abstracting the notions of monotonicity and acknowledgement (developed on the implementation state graph in Appendix B) to specification states and transitions in such a way that they can be checked through inspection of the cube approximation. Since the cube approximation is a conservative estimate of the reachable state space, the conditions are sufficient, and not necessary, conditions for correctness.

Consider first the monotonicity of a particular internal or primary output signal u. Since only fanins of u can affect $f_u(q)$ and since u's external evaluation is the same in all implementation states that project onto a specification state s, to abstract the notion of monotonicity to specification states only the effects of each fanin w of a signal u in each specification state s need be considered. We present three sufficient conditions which each guarantee that a fanin w can not cause $f_u(q)$ to be non-monotonic during any implementation state transition from a state $q \in Q_{\textit{CEA}}$ that projects onto s.

The first condition is when fanin w *encourages* the expected change on u, that is when w helps the signal u reach its external evaluation. For example, as depicted in Figure 13(b), the fanin w_3, which is supposed to rise, is an encouraging fanin of an AND gate that is supposed to rise. The rationale behind this is that from core externally-aligned states we know w_3 can only change in the direction of its external evaluation, i.e., it can only rise. It is easy to see that w_3 rising cannot cause u to be non-monotonic. This characteristic of encouraging fanins is made explicit in the following definition,

$$\begin{aligned}&\textit{encouraging}(w,u,s) \equiv w \in \textit{FI}(u) \wedge \\ &\quad \forall q \in \mathcal{B}^{A_{\textit{Impl}}}\left[\left[q(w) \neq \textit{ext_eval}(s)(w) \ \wedge \ f_u(q) = \textit{ext_eval}(u)(s)\right] \Rightarrow \left[f_u(\textit{bitcomp}(w,q)) = \textit{ext_eval}(u)(s)\right]\right].\end{aligned}$$

stable until a new request is issued to the signal. Using the notion of stability provided by the cube approximation, we present sufficient conditions that test monotonicity and acknowledgement of all internal signals and primary outputs in all implementation state transitions.

C.1 Mapping the cube approximation to a set of states

A cube by itself does not correspond to a set of implementation states since a cube does not define the value of the external circuit signals. However, given a reference specification state s, the function $\mathit{CtoQ}(c, s)$ maps a cube c to a set of implementation states in the traditional way.

A cube approximation of an implementation state graph consists of several cubes: one for each state in the specification state graph (denoted $\mathit{CA}(s)$ and called a *state cube*) and one for each transition in the specification state graph (denoted $\mathit{CA}(s, v)$ and called a *transition cube*). If only the state cubes are considered, then the set of implementation states represented by a cube approximation is

$$Q_{EA} \cap \bigcup_{s \in \Phi} \mathit{CtoQ}(\mathit{CA}(s), s).$$

The intersection with Q_{EA}, above, is necessary to avoid grossly overestimating Q_{CEA}. Since every state in Q_{CEA} is externally-aligned, we know that intersection with Q_{EA} does not cause any states to be missed.

Even when Q_{EA} is used in this way, however, the resulting approximation is often too conservative to be useful. The transition cubes $\mathit{CA}(s, v)$ are used to further constrain the approximation. Let q be an externally-aligned implementation state such that $\mathit{proj}(A_{Spec})(q) = s$. The state q is in the cube approximation iff

- $q \in \mathit{CtoQ}(\mathit{CA}(s), s)$ and
- for all primary outputs $v \in O$, if v is enabled in q, then $q \in \mathit{CtoQ}(\mathit{CA}(s, v), s)$.

In other words, q is in the cube approximation iff $q \in Q_{CA}(s)$, where

$$Q_{CA}(s) \;=\; Q_{EA} \;\cap\; \mathit{CtoQ}(\mathit{CA}(s), s) \;\cap\; \left\{ q \in \mathcal{B}^{A_{Impl}} \mid \forall v \in O[\mathit{enabled}(v, q) \;\Rightarrow\; q \in \mathit{CtoQ}(\mathit{CA}(s, v), s)] \right\}.$$

The set of all implementation states represented by the cube approximation is

$$Q_{CA} = \bigcup_{s \in \Phi} Q_{CA}(s).$$

C.2 Required fanins and acknowledgement

We say a fanin is *required* by an output in a specification state given a corresponding cube if in the specification state the firing of the fanin must precede the firing of the output. This subsection develops a formalization of this intuition and uses it to define the notion of *acknowledgement*. Acknowledgement, in turn, is used to determine the stability of internal signals in computing the cube approximation.

Given a specification state s, a cube c, and a signal u, all implementation states in $\mathit{CtoQ}(c, s)$ in which u's internal evaluation equals its external evaluation are considered. A fanin w of u is required if in all of these states it is at its settled value as defined below.

$$\begin{aligned} &\mathit{required}(w, u, s, c) \equiv \\ &\quad w \in \mathit{FI}(u) \;\wedge\; \forall q \in \mathit{CtoQ}(c, s) \left[\left[f_u(q) = \mathit{ext_eval}(s)(u) \right] \Rightarrow \left[q(w) = \mathit{extend}(s)(w) \right] \right]. \end{aligned}$$

For example, the output of an AND gate that is supposed to rise in state s cannot rise until all its fanins rise. Hence, all the fanins of this AND gate in state s are required fanins. Similarly, if all but one fanin of an AND gate that is supposed to fall in state s are stable high, then the AND gate cannot fall until that last fanin of the AND gate falls. Hence, in state s, the last fanin is a required fanin of the falling AND gate.

Informally, given a specification state s and a primary output v which is enabled in s we say the corresponding state transition $s \xrightarrow{v} s'$ *acknowledges* an internal signal if the internal signal must settle to its external evaluation before the state transition fires. As described above, this happens when w is a required fanin of some signal u whose internal evaluation must reach its external evaluation before the enabled primary output v fires. This condition on w holds in two significant cases. First, it holds when u is identically the primary output v that is enabled to fire

q, $f_u(q)$ equals u's external evaluation, then in q', $f_u(q)$ should also equal u's external evaluation. We denote this condition by *monotonic*(u, q, q').

Now consider the boundary of a signal region of u. That is, consider a transition $q \xrightarrow{v} q'$ in which the external evaluation of u changes. In this case, u should be at its external evaluation in q, which means that u should complete the 0-to-1 or 1-to-0 change in value before the circuit leaves the region. Otherwise, there exists a sequence of states in which before v fires u is enabled, and after v fires u is disabled, either directly from v firing or from the firing of signals subsequently enabled. In other words u must be *acknowledged* before its external evaluation changes and denote this condition is denoted by *acknowledged*(u, q, q').

Monotonicity and acknowledgement of all internal signals and primary outputs combined are necessary and sufficient conditions for a complex-gate equivalent circuit to be hazard-free [4, 3]. However, to directly prove that our synthesis algorithm generated circuits that satisfy these conditions would be very difficult. Instead, we use these conditions as the foundation of sufficient conditions for correctness that are more useful in terms of proving our synthesis procedure correct.

B.2 Relationship between monotonicity/acknowledgment and successful implementation states

It is useful to identify the relationship between monotonicity and acknowledgement and the set of successful implementation states. Monotonicity and acknowledgement together ensure that enabled signals are never at their external evaluation, that is, they are enabled to change only in the direction of their external evaluation. These conditions are necessary conditions for correctness because the existence of such a reachable implementation state guarantees that the circuit is hazardous. These necessary and sufficient conditions imply that it is possible to exactly check correctness by analyzing a subset of reachable implementation states.

To formalize this a signal u is *externally aligned* in q if in q the signal u is not both enabled and at its external evaluation, i.e.,

$$\textit{externally_aligned}(u,q) \quad \equiv \quad \neg \left[\textit{enabled}(u,q) \wedge \left[q(u) = \textit{ext_eval}(\textit{proj}(A_{\textit{Spec}})(q))(u)\right]\right] .$$

An implementation state q is externally aligned if all signals are externally aligned in q, and the set of externally aligned states in $\mathcal{B}^{A_{\textit{Impl}}}$ is denoted $Q_{\textit{EA}}$. The failure state $q_{\textit{fail}}$, by definition, is not externally aligned.

The subset of externally-aligned states reachable through externally-aligned states, referred to as the *core set of externally-aligned states* and denoted by $Q_{\textit{CEA}}$, can be defined as the smallest set of states satisfying:

$$\textit{extend}(s_0) \in Q_{\textit{CEA}} \vee$$

$$\left[\exists q \in Q_{\textit{CEA}}\ \forall u \in N \cup O \left[\textit{hazard-free}(u,q,q') \ \wedge\ \textit{externally_aligned}(u,q')\right]\right] \ \Rightarrow\ \left[q' \in Q_{\textit{CEA}}\right] .$$

We use the adjective "core" to describe this set to emphasize that there may be externally-aligned states outside $Q_{\textit{CEA}}$, i.e., those reachable only by sequences of states that contain externally-misaligned states. We have shown that for all correct circuits $Q_{\textit{CEA}}$ equals $Q_{\textit{Succ}}$ [3]. However, for some incorrect circuits $Q_{\textit{CEA}}$ is a subset of $Q_{\textit{Succ}}$.

In summary, $Q_{\textit{CEA}}$ is a subset of reachable states from which it is possible to check for hazard-freedom. The check is simply to see if any state transitions from this set violate monotonicity or acknowledgement. This allows us to focus on analyzing $Q_{\textit{CEA}}$ instead of the potentially larger set $Q_{\textit{Reach}}$ to determine hazard-freedom. As shown in the next two appendices, this result is very useful in developing sufficient conditions for correctness.

C The cube approximation

A cube approximation is essentially a state graph with cubes that annotate specification states and transitions. Each cube describes the approximate behavior of all internal signals for the particular state or transition the cube is associated with. For example, if in a state cube c associated with state s, $c(u) = X$, then the cube approximation states that internal signal u can be changing when the circuit is in specification state s. The cube approximation compactly represents a large set of implementation states.

This appendix introduces an efficient algorithm to find the best cube approximation that represents a superset of important implementation states (i.e., $Q_{\textit{CEA}}$), developed in Appendix B. This algorithm relies on a formulation of the notion of *acknowledges* in which transitions of output signals allow us to infer the stability of internal signals. The general idea is that once an output signal transition acknowledges a request on a signal, the signal remains

gate-level circuits composed of simple gates in which hazards can occur as a result of the added modeled delay within the circuit. Hence, the second part of our notion of correctness is *hazard-freedom*,

Hazard-freedom is a safety property of the actual behavior of a circuit implementation in a particular environment. Their joint behavior is modeled using an *implementation state graph*. An implementation state graph is defined by $\langle Q_{Reach}, R, q_0 \rangle$, where Q_{Reach} is the set of reachable implementation states, R is a state transition relation, and q_0 is the initial state.

The initial state of the implementation q_0 is defined to be the implementation-state extension of the initial specification state s_0 (i.e., $q_0 = \mathit{extend}(s_0)$). This model is based on the assumption that after circuit power-up the environment holds the external signals fixed until all internal signals have time to settle.

The transition relation R includes one transition for every enabled signal in every implementation state. The destination states of these transitions depend upon whether or not the transition is hazard-free. A transition associated with an enabled signal v in successful state q is defined to be *hazard-free* on u if the firing of v does not disable u. That is,

$$\mathit{hazard\text{-}free}(u, q, q') \equiv u = v \ \vee \ \left[\mathit{enabled}(u, q) \Rightarrow \mathit{enabled}(u, q')\right],$$

where $q' = \mathit{bitcomp}(v, q)$. Note that we apply this definition only to internal and output signals, not to primary inputs which are allowed to disable other inputs.

The state the circuit enters when an enabled signal fires depends on whether the transition causes a *hazard* on any gate output u (internal signal or primary output). If the firing of the signal causes a hazard at any gate output, then the state entered is defined to be the failure state q_{fail}. Otherwise, the destination state of the transition is $\mathit{bitcomp}(v, q)$. Implementation state transitions are denoted by $q \xrightarrow{v} q'$ similar to specification state transitions. In addition, if the signal that changes, v, is not relevant, the notation $q \rightarrow q'$ may be used. This model is based on the assumption that an internal hazard in a speed-independent circuit can always propagate to a primary output and hence always represents a circuit malfunction. This assumption, in general, might be conservative, but has been proven true for a large class of circuits [7].

The set of reachable states of an implementation SG, Q_{Reach}, can be recursively defined as follows:

$$\mathit{extend}(s_0) \in Q_{Reach};$$

$$\left[\exists q \in Q_{Reach} \ \left[q \rightarrow q'\right]\right] \ \Rightarrow \ \left[q' \in Q_{Reach}\right].$$

The set of successful states Q_{Succ} is the set of reachable states minus q_{fail}.

A circuit is hazard-free if the failure state is not reachable, i.e.,

$$q_{fail} \notin Q_{Reach}.$$

Notice that hazard-freedom by itself is not a sufficient check for correctness because circuits that do not behave as specified may still be hazard-free.

B Necessary and sufficient conditions for correctness

In section 3.2, we gave intuitive explanations of the notions of acknowledgement and monotonicity which we claimed were necessary and sufficient conditions for correctness. This section formalizes these claims.

B.1 Formalization of monotonicity and acknowledgement

To formalize the notions of monotonicity and acknowledgement consider a sequence of implementation states $q_1, \ldots, q_i, \ldots, q_n$ that project onto specification states within a signal region for signal u. In each of these states u, the signal is requested to settle to its external evaluation. However, the request does not reach the internal signal until an implementation state is reached in which the signal's local evaluation equals its external evaluation, i.e.,

$$f_u(q_i) = \mathit{ext_eval}(\mathit{proj}(A_{Spec})(q_i))(u). \tag{1}$$

Once this implementation state is reached the gate associated with u, u is driven to its new external evaluation after some delay. In a correct circuit, once this implementation state is reached, the request is maintained until after the circuit leaves the signal region. In other words, the predicate (1) *monotonically* increases from false to true through the sequence. For example, consider a state transition $q \rightarrow q'$ within the sequence of implementation states. If in

A Definition of Correctness

Informally, a correct speed-independent circuit is one whose behavior satisfies a given specification under all combinations of gate delays. We formalize this notion of satisfies with a definition of correctness of speed-independent circuits that is comprised of two parts: complex-gate equivalence which primarily deals with functional correctness and hazard-freedom which primarily deals with behavioral correctness, i.e., transient behavior.

A.1 Complex-gate equivalence

The definition of complex-gate equivalence is based on a couple of ideas. The status of a speed-independent circuit at any point in time is completely characterized by an *implementation state*, that is either a bitvector q over A_{Impl}, (i.e., $q \in 2^{A_{Impl}}$), called a *successful implementation state* or the special value, q_{fail}, that models the failure state of an implementation entered after the occurrence of a hazard (described later in this Appendix). An internal signal or primary output u is *enabled* in an implementation state q if u's value does not equal $f_u(q)$. A primary input u is enabled in an implementation state q if u is enabled in the specification state on which it projects, i.e., the specification state s in which for all u in $A_{Spec}, s(u) = q(u)$, denoted $proj(A_{Spec})(q)$. No signal is enabled in the failure state q_{fail}. Summarizing we have,

$$enabled(u,q) \equiv \begin{cases} f_u(q) \neq q(u) & u \in N \cup O \ \wedge \ q \neq q_{fail} \\ enabled(u, proj(A_{Spec})(q)) & u \in I \ \wedge \ q \neq q_{fail} \\ FALSE & q = q_{fail}. \end{cases}$$

For each specification state s in an externally-cut circuit, we claim that there exists a unique implementation state $extend(s)$, called an *implementation-state extension*, that projects onto s and in which no internal signals are enabled. The value of signal u in $extend(s)$ is called its *settled value*. The uniqueness of $extend(s)$ can be easily proven since when a circuit is externally-cut, the circuit becomes acyclic and combinational.

For each specification state s, the value that a signal is driven to in s is called the the *external evaluation* of the signal in s. The external evaluation of an internal or primary output u in state s equals the local evaluation of u in the implementation-state extension of s. On the other hand, the external evaluation of an input signal u in s equals $s(u)$ if the input is not enabled in s and $\overline{s(u)}$ if u is enabled in s. Summarizing, the external evaluation for each signal in a state s is given by $ext_eval(s)$ defined as follows

$$ext_eval(s)(u) \equiv \begin{cases} f_u(extend(s)) & \text{if } u \in N \cup O \\ s(u) & \text{if } u \in I \text{ and not } enabled(u,s) \\ \overline{s(u)} & u \in I \text{ and } enabled(u,s). \end{cases}$$

A circuit is *complex-gate equivalent* to its specification when the external evaluation of all primary outputs agree with the specification, that is, if the external evaluation of each primary output differs from its current value in exactly those specification states in which it is enabled, i.e.,

$$\forall s \in \Phi \ \forall u \in O \left[\left[ext_eval(s)(u) \neq s(u) \right] \Leftrightarrow enabled(u,s) \right].$$

Intuitively, complex-gate equivalence states that the specification and implementation should behave the same when considering only external signals and *ignoring hazards*. Complex-gate equivalence is both a safety and a liveness property. It is a safety property because it ensures that ignoring internal signals the circuit behavior is allowed by the specification. It is a liveness property because it ensures that the circuit can exhibit any specified behavior given the appropriate input choices and gate delays. It is similar to the liveness notion *completeness with respect to specification* introduced by Ebergen [19]

A.2 Hazard-freedom

If each primary output is built using a single atomic complex-gate, then complex-gate equivalence is the only correctness criterion needed since under these conditions there are no hazards. However, this paper deals with

[44] J. Rho, G. Hachtel, F. Somenzi, and R. Jacoby. Exact and heuristic algorithms for the minimization of incompletely specified state machines. *IEEE Transactions on Computer-Aided Design*, pages 167–177, February 1994.

[45] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, University of California, Berkeley, May 1992.

[46] P. Siegel and G. DeMicheli. Decomposition methods for library binding of speed-independent asynchronous designs. In *IEEE/ACM 1994 ICCAD Digest of Technical Paper*, 1994. To appear.

[47] J. A. Tierno, A. J. Martin, D. Borković, and T. K. Lee. A 100-MIPS GaAs asynchronous microprocessor. *IEEE Design & Test of Computers*, 11(2):43–49, 1994.

[48] S. H. Unger. *Asynchronous Sequential Switching Circuits*. New York: Wiley-Interscience, 1969. (re-issued by R. E. Krieger, Malabar, 1983).

[49] C.H. (Kees) van Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, and F. Saeijs. A fully-asynchronous low-power error corrector for the digital compact cassette player. In *IEEE International Solid-State Circuits Conference*, 1994.

[50] K. van Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *Proc. European Conference on Design Automation (EDAC)*, pages 384–389, 1991.

[51] P. Vanbekbergen, B. Lin, G. Goossens, and H. de Man. A generalized state assignment theory for transformations on signal transition graphs. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 112–117. IEEE Computer Society Press, November 1992.

[52] V. I. Varshavky, editor. *Self-Timed Control of Concurrent Processes*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1990.

[23] S. Jeong and F. Somenzi. A new algorithm for the binate covering problem and its application to the minimization of boolean relations. In *IEEE ICCAD Digest of Technical Papers*, pages 417–420, 1992.

[24] A. Kondratyev, M. Kishinevsky, B. Lin, P. Vanbekbergen, and A. Yakovlev. *private communication*, July 1993.

[25] A. Kondratyev, M. Kishinevsky, B. Lin, P. Vanbekbergen, and A. Yakovlev. Basic gate implementation of speed-independent circuits. In *Proc. ACM/IEEE Design Automation Conference*, 1994.

[26] L. Lavagno. *Synthesis and Testing of Bounded Wire Delay Asynchronous Circuits from Signal Transition Graphs*. PhD thesis, University of California, Berkeley, 1992.

[27] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli. Algorithms for synthesis of hazard-free asynchronous circuits. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, 1991.

[28] L. Lavagno, C. Moon, R. Brayton, and A. Sangiovanni-Vincentelli. Solving the state assignment problem for signal transition graphs. In *Proc. ACM/IEEE Design Automation Conference*, pages 568–572. IEEE Computer Society Press, June 1992.

[29] B. Lin, O. Coudert, and J. C. Madre. Symbolic prime generation for multiple-value functions. In *Design Automation Conference*, pages 40–44. ACM/IEEE, June 1990.

[30] K.-J. Lin and C.-S. Lin. Direct synthesis of hazard-free asynchronous circuits from STGs based on lock relation and MG-decomposition approach. In *Proc. European Conference on Design Automation (EDAC)*, pages 178–183. IEEE Computer Society Press, 1994.

[31] A. Marshall, B. Coates, and P. Siegel. Designing an asynchronous communications chip. *IEEE Design & Test of Computers*, 11(2):8–21, 1994.

[32] A. J. Martin. Programming in VLSI: from communicating processes to delay-insensitive VLSI circuits. In C.A.R. Hoare, editor, *UT Year of Programming Institute on Concurrent Programming*. Addison-Wesley, 1990.

[33] A. J. Martin. *Private Communication*, October 1994. A. J. Martin is a professor of computer science at Caltech.

[34] A. J. Martin, S. M. Burns, T. K. Lee, D. Borković, and P. J. Hazewindus. The design of an asynchronous microprocessor. In *Decennial Caltech Conference on VLSI*, pages 226–234, 1989.

[35] E. J. McCluskey. *Logic Design Principles with Emphasis on Testable Semicustom Circuits*. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[36] P. C. McGeer and R. K. Brayton. *Integrating Functional and Temporal Domains in Logic Design*. Kluwer Academic Publishers, Norwell, MA, 1991.

[37] T. H.-Y. Meng, R. W. Brodersen, and D. G. Messershmitt. Automatic synthesis of asynchronous circuits from high-level specifications. *IEEE Transactions on Computer-Aided Design*, 8(11):1185–1205, November 1989.

[38] R. E. Miller. *Switching Theory, Volume II: Sequential Circuits and Machines*. Wiley, New York, 1965.

[39] D. E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *Proceedings of an International Symposium of the Theory of Switching*, pages 204–243, 1959.

[40] C. J. Myers and T. H.-Y. Meng. Synthesis of timed asynchronous circuits. *IEEE Transactions on VLSI Systems*, 1(2):106–119, June 1993.

[41] C. J. Myers, T. G. Rokicki, and T. H.-Y. Meng. Automatic synthesis of gate-level timed circuits with choice. To appear at 1995 Chapel Hill Conference on Advanced Research in VLSI.

[42] T. Nanya, Y. Ueno, H. Kagotani, M. Kuwako, and A. Takamura. TITAC: Design of a quasi-delay-insensitive microprocessor. *IEEE Design & Test of Computers*, 11(2):50–63, 1994.

[43] S. M. Nowick. *Automatic Synthesis of Burst-Mode Asynchronous Controllers*. PhD thesis, Stanford University, Department of Computer Science, 1993.

[2] P. A. Beerel. *CAD Tools for the Synthesis, Verification, and Testability of Robust Asynchronous Circuits*. PhD thesis, Stanford University, August 1994.

[3] P. A. Beerel, J. R. Burch, and T. H.-Y. Meng. Efficient verification of speed-independent circuits, May 1994. Submitted for publication in *IEEE Transactions on Computer-Aided Design*.

[4] P. A. Beerel, J. R. Burch, and T. H.-Y. Meng. Sufficient conditions for correct gate-level speed-independent circuits. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*, November 1994.

[5] P. A. Beerel and T. H.-Y. Meng. Automatic gate-level synthesis of speed-independent circuits. In *IEEE 1992 ICCAD* Digest of Technical Papers, pages 581–586, 1992.

[6] P. A. Beerel and T. H.-Y. Meng. Gate-level synthesis of speed-independent asynchronous control circuits, 1992. In collection of papers of the *ACM International Workshop on Timing Issues in the Specification of and Synthesis of Digital Systems*.

[7] P. A. Beerel and T. H.-Y. Meng. Semi-modularity and testability of speed-independent circuits. *INTEGRATION, the VLSI journal*, 13(3):301–322, September 1992.

[8] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic, 1984.

[9] R. K. Brayton and F. Somenzi. An exact minimizer for boolean relations. In *International Conference on Computer-Aided Design*, pages 316–320. IEEE Computer Society Press, 1989.

[10] W. O. Budde, H.-G. Keller, H.-J. Reumerman, and P. van de Wiel. An asynchronous, high-speed packet switching component. *IEEE Design & Test of Computers*, 11(2):33–42, 1994.

[11] S. Burns. *Performance Analysis and Optimization of Asynchronous Circuits*. PhD thesis, California Institute of Technology, 1991.

[12] T.-A. Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-theoretic Specifications*. PhD thesis, Massachusetts Institute of Technology, 1987.

[13] T.-A. Chu. Synthesis of hazard-free control circuits from asynchronous finite state machine specifications. *Journal of VLSI Signal Processing*, 7(1/2):61–84, February 1994.

[14] T.-A Chu and N. S. Mani. CLASS: A CAD system for automatic synthesis and verification of asynchronous finite state machines. *Integration, the VLSI journal*, 15(3):263–289, October 1993.

[15] B. Coates, A. Davis, and K. Stevens. The Post Office experience: Designing a large asynchronous chip. *Integration, the VLSI journal*, 15(3):341–366, October 1993.

[16] M. Damiani and G. De Micheli. *Don't Care* Set Specification in Combinational and Synchronous Logic Circuits. Technical Report CSL-TR-92-531, Stanford University, 1990.

[17] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, Inc., New York, New York, 1994.

[18] D. L. Dill. Trace theory for automatic hierarchical verification of speed-independent circuits. ACM Distinguished Dissertations, 1989.

[19] J. C. Ebergen. A verifier for network decompositions of command-based specifications. In *Proc. of the 26th Annual HICSS*, pages 310–318. IEEE Computer Society Press, 1993.

[20] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. A micropipelined ARM. In *VLSI '93*, 1993.

[21] A. Grasselli and F. Luccio. A method for minimizing the number of internal states in incompletely specified sequential networks. *IEEE Transactions on Electronic Computers*, pages 350–359, June 1965.

[22] A. Grasselli and F. Luccio. Some covering problems in switching theory. In G. Biorci, editor, *Network and Switching Theory*. Academic Press, 1966.

we attempt to decompose the optimized circuit and, if this fails, we decompose the unoptimized circuit and then perform optimizations.

Our current algorithm for breaking up the high-fanin gates is computationally expensive primarily due to the number of decompositions that it explores. Decomposing the high-fanin gates in the largest circuit *pe-send-ifc* took approximately 20 minutes on a SPARC 20 with 128 Megabytes of RAM. Further research is necessary to find more computationally efficient methods for decomposition. The synthesis run-time excluding the decomposition step, however, is small, taking under 1 minute for all our examples.

Our final results, depicted in Table 4, demonstrate that decomposition of gates into multi-level logic is not only feasible, but often decreases circuit area. The primary reason for this is that the breaking up of high-fanin gates increased the amount of logic sharing between logic blocks. These results suggest that multi-level logic optimizations may be feasible and worthwhile in speed-independent circuits.

8.2 Comparison with SIS

We compare our synthesis algorithm with an alternative design technique for asynchronous circuits in which synchronous techniques are used to develop the initial logic, and delay lines are then added to remove hazards [27, 26]. For all circuits but *pe-send-ifc* the results are obtained using an optimal delay insertion algorithm [26] incorporated into Berkeley's SIS, with the minimum delay of all gates set to 0.8 times their maximum gate delay. For *pe-send-ifc*, the largest of the benchmark circuits, the optimal algorithm did not complete after 3 days, and the reported result is obtained with a much faster greedy algorithm [27].

The synthesis results shown in Table 5 indicate that our optimized circuits are, on average, 14% larger but 13% faster than those obtained with SIS. The reasons for our speed advantage are two-fold. First, the recommended optimization script that is used with SIS seems to optimize for area rather than speed, and secondly, the delay lines that SIS needs to add to ensure hazard-freedom represents a significant fraction of the overall circuit delay, whereas a speed-independent circuit would by definition eliminate these unnecessary delay lines. We note that current research is directed towards technology mapping for speed-independent circuits which may significantly improve our results [46].

8.3 Summary

This paper describes a provably correct synthesis procedure from SG specifications to speed-independent circuit implementations using only limited-fanin basic gates. These SG specifications can easily be derived from many high-level languages and are capable of specifying a wide class of circuits including many examples from academia and industry. Our synthesis procedure first produces a standard C-implementation using binate covering formulations designed for both single-cube and multi-cube covers. In addition, our procedure incorporates an efficient methodology for logic transformations that is based on observability don't cares and incremental verification. This work represents the first use of multi-level logic in the domain of gate-level speed-independent circuits. Finally, our procedure has been fully automated into a CAD tool, and it has been successfully applied to many examples resulting in circuits that are on average 13% faster with only 14% area penalty when compared with the most comprehensive benchmark set available [27],

Acknowledgments

We would like to thank Professor Luciano Lavagno of Politecnico University, Torino, Italy for creating the benchmark of specifications and for his cooperation in obtaining an up-to-date fair comparison between our two methods. We would also like to acknowledge Dr. Jerry R. Burch of Cadence Berkeley Labs for his invaluable collaboration on the formal verification results that are tightly related to this work. In addition, we greatly appreciate many valuable comments on earlier versions of this paper that we received from Dr. Polly Siegel of HP Labs. Finally, we are very grateful to Professor David Dill of Stanford University for his steady support and guidance.

References

[1] D. B. Armstrong, A.D. Friedman, and P. R. Menon. Design of asynchronous circuits assuming unbounded gate delays. *IEEE Transactions on Computers*, C-18(12):1110–1120, December 1969.

Our Synthesis CAD Tool vs. SIS			
Circuit	$\mid \Phi \mid$	Our CAD tool Area/Del	SIS Area/Del
chu133	24	232 / 4.8	240 / 5.6
chu150	26	248 / 4.8	232 / 5.8
chu172	12	168 / 3.6	104 / 1.6
converta	18	376 / 6.0	384 / 6.2
ebergen	18	352 / 4.8	216 / 3.2
full	16	112 / 2.4	208 / 5.8
hazard	12	248 / 4.8	200 / 4.2
hybridf	80	152 / 2.4	284 / 7.4
nowick	20	456 / 6.0	232 / 4.6
alloc-outbound	20	400 / 6.0	272 / 4.0
mp-forward-pkt	22	320 / 3.6	232 / 3.4
nak-pa	58	336 / 5.4	256 / 3.4
pe-rcv-ifc	53	1304 / 7.6	816 / 8.6
pe-send-ifc	117	1632 / 9.4	1520 / 14.6
ram-read-sbuf	39	432 / 5.0	416 / 6.6
rcv-setup	11	144 / 2.4	144 / 4.0
sbuf-ram-write	64	320 / 4.8	296 / 4.0
sbuf-read-ctl	19	296 / 4.8	272 / 4.2
sbuf-send-ctl	27	280 / 4.8	280 / 3.2
sbuf-send-pkt2	26	504 / 6.0	416 / 8.0
sendr-done	9	80 / 3.6	112 / 4.2
qr42	18	352 / 4.8	216 / 3.2
rpdft	22	224 / 6.0	264 / 7.6
trimos-send	336	648 / 4.8	576 / 4.6
vbe10b	256	792 / 6.0	832 / 11.8
vbe5b	24	192 / 4.8	224 / 5.2
vbe5c	24	200 / 3.6	192 / 4.2
wrdatab	216	784 / 4.8	712 / 7.2
Totals		11584 / 137.8	10148 / 156.4

Table 5: Comparative synthesis results.

The synthesis algorithm and optimization techniques described have been automated in a CAD tool written in C and tested on a large benchmark of circuits from academia and industry. To estimate the performance of our synthesis algorithm, we have generated a speed-independent limited-fanin gate-level implementation for all the specification examples taken from the benchmarks given in [26]. The SG specifications for these examples have been obtained from their STG specifications using a modified version of the Berkeley's synthesis system SIS [45].

8.1 Breaking up complex and high-fanin gates

To estimate the advantages of multi-level logic and the cost of our break-and-repair decomposition methodology, we compared the area before and after the addition of acknowledgement wire forks. For the original circuits, we needed AND gates with five to eight inputs and a few AND-OR-INVERT gates. We used the SIS asynchronous library as a basis of area delays and extrapolated the area of the high-fanin gates from the smaller gates in the library.

We experimented with decomposing high-fanin gates before and after performing logic optimization. In our experience, decomposing gates after optimizations produces better circuits, but sometimes is not possible. Optimizations, such as removing C-elements and merging gates, makes it more difficult to find good acknowledgement wire forks. As a result, the high-fanin gates in the optimized *pe-send-ifc* could not be decomposed. For all other circuits, however, decomposing the circuits after optimization is successful and produces better results. Hence,

Breaking up high-fanin and complex gates					
				Area comparison	
Circuit	# complex-gates	# high-fanin gates	# awfs	Before	After
var1	1	-	0	264	264
xyz	1	-	0	192	192
trimos-send	3	-	3	648	648
MUXcondSI	-	2	2	648	576
MUXSI	-	3	2	824	784
tsbmSI	-	2	2	960	952
vbe10b	-	1	0	792	784
pe-rcv-ifc	-	2	4	1320	1304
pe-send-ifc	-	4	13	1600	1632
Totals				7248	7136

Table 4: Results for breaking up high-fanin and complex gates.

implementation states. Our algorithm efficiently finds the best cube approximation that represents a superset of Q_{CEA}.

We developed conditions—abstract versions of acknowledgment and monotonicity—which can be tested directly on the cube approximation to verify correctness. The conditions are sufficient and not necessary because the cube approximation is only a conservative estimate of the reachable state space.

7.3.2 Localizing the effects of a logic transformation

Our verification procedure determines if a logic transformation is non-hazard-increasing by recalculating the cube approximation and testing the sufficient conditions for correctness. Fortunately, both our algorithm for finding the cube approximation and our sufficient conditions for correctness facilitate incremental analysis.

First, we localize the effects of a particular logic transformation in a circuit. If we consider the underlying undirected graph modeling the circuit that is cut by all external signals, the effects of a logic transformation are guaranteed to be localized to only the connected components of gates involved in the transformation. In larger circuits it is not uncommon that each output signal be implemented with a separate non-overlapping cone of logic. In this case, each cone of logic represents a separate connected component and our analysis may be limited to a small portion of the circuit. Of course, if the circuit is a single connected component, then this localization strategy does not decrease the needed computation. In fact, in this degenerate case, the computation required is slightly larger due to the time needed to find the connected component (linear with respect to the size of circuit).

Once the affected connected components are identified, only the corresponding portions of the cube approximation need be re-computed. This is achieved with the same algorithm described in [3] with the only variant being different initial conditions of the cube approximation. Finally, our procedure tests our abstract acknowledgement and monotonicity conditions on only the gates in the affected connected components.

7.4 Accepting logic transformations

If a logic transformation fails the ODC check, then it is not complex-gate-equivalence preserving and is rejected. Otherwise, the stability information is updated and hazard-freedom is checked. Note that for the case of repairing decompositions and adding connections as acknowledgement wire forks, the circuit need not satisfy all the verification conditions to be accepted. Since many subregions may need to be acknowledged, many acknowledgement wire forks may be needed before the circuit is repaired. Therefore, the procedure accepts an acknowledgement wire fork if it repairs the violations in the subregion being analyzed and does not introduce new violations. Once a logic transformation is accepted, the procedure updates the external evaluations, stores the new stability information, and recalculates those ODC sets that may have changed.

8 Results and conclusions

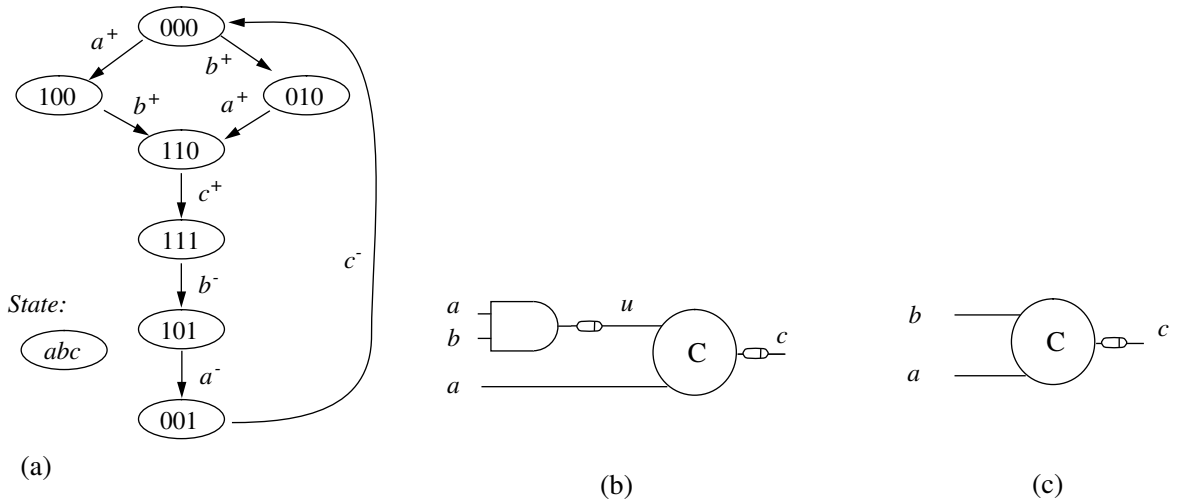


Figure 12: (a) A SG specification, (b) the unoptimized circuit, and (c) the optimized circuit

7.2.5 An example

As an example of a logic transformation using ODCs, consider the specification and unoptimized circuit obtained from the techniques described in Section 3 shown in Figure 12. Using Algorithm 7.1, we obtain $ODC(u) = \{[000], [010], [111], [101]\}$. Removing the connection from a to u, our ODC check is satisfied and also preserves hazard-freedom. After removing the resulting non-inverting buffer, the optimized circuit is as given in Figure 12(c).

7.3 Incrementally verifying hazard-freedom

To determine if a logic transformation introduces hazards in a circuit, we use the theory described in [3, 4] which develops a tight approximation to the reachable state space and provides a corresponding tight set of sufficient conditions for correctness. The major result from [3] is an algorithm to test these sufficient conditions that is linear in time with respect to the size of the circuit and state graph for a broad class of circuits, exponentially faster than most verification tools. In this subsection, we explain how this theory and algorithm can facilitate incremental verification, thereby allowing a fast check that logic transformations are non-hazard-increasing.

7.3.1 The cube approximation and sufficient conditions for correctness

The theoretical foundation of our verification procedure is the necessary and sufficient conditions for correctness, acknowledgement and monotonicity, described in Section 3.2. The notions of acknowledgement and monotonicity lead to the identification of a core set of reachable implementation states, referred to as Q_{CEA}, from which hazard-freedom can be verified. The computational efficiency of our verification procedure is achieved through approximating (finding a superset of) Q_{CEA}. The key result is an approximation that is tight in practice and is exponentially easier to find than identifying Q_{CEA} itself. The algorithm checks the conditions on this approximation that guarantee the circuit satisfies acknowledgement and monotonicity.

The intuition behind our approximation is the notion of a stable internal signal, i.e., an internal signal that is guaranteed to have reached its external evaluation. The key observation is that an internal signal is stable between the time it is acknowledged and the next request for it to change is issued. Our algorithm uses a data structure, called a *cube approximation*, that captures this information in a compact form. Essentially, the cube approximation is a state graph that has been annotated with stability information regarding all internal signals. Specifically, each state and state transition is annotationed with a *cube* that designates if each internal signal is stable in the particular state or transition. There exists a well-defined correspondence between a cube approximation and a set of reachable

Algorithm 7.1 (Find Observability Don't Cares)
Find_ODC(G, M) {
  $D = \{$ *primary outputs* $\}$
  initialize ODC$(u) = \emptyset$ *for all* $u \in A_{Impl}$
  while $(D \neq A_{Impl})$
    select $u \in \{A_{Impl} - D\}$ *such that FO*$(u) \subseteq D$
    foreach *state* $s \in \Phi$
      if (*for at most one* $v \in$ *FO*(u) *is sensitized*$(u, v, s,$ *CA*$(s))$
        and (*for all* $v \in$ *FO*(u) *does* (*sensitized*$(u, v, s,$ *CA*$(s))$
        and $s \in$ *ODC*(v)) *or insensitized*$(u, v, s,$ *CA*$(s))))$
      then *ODC*$(u) =$ *ODC*$(u) \cup \{s\}$
    $D = D \cup \{u\}$
}

Figure 11: Algorithm for finding observability don't cares

Consider the path [010] → [011] → [001] → [101] inside the rising request of x. In states [010] the signal u tends to enable x to rise. But, u can fall as soon as state [011] is entered. At the same time, if the non-inverting buffer attached to v is slow, v can be still rising. This can cause x to glitch in state [011] as illustrated in Figure 10(c). This hazard does not occur in the circuit shown in Figure 10(b) because in that circuit u could not fall until v has risen. For this reason, we conclude that [011] should not be included in the set *ODC*(u).

This distinction between asynchronous and synchronous circuits is captured by redefining the notions of sensitization for asynchronous circuits. We provide informal definitions here and formalize them in Appendix D. A fanin w is an insensitized fanin of signal u in an state s if f_u is completely determined by the value of w in state s. For example, an input to an OR gate u is a sensitized fanin of u if and only if all side-inputs are *stable* 0. A fanin w is an insensitized fanin of signal u in an state s if f_u is independent of the value of w in state s. For example, an input to an OR gate u is an insensitized fanin u if and only if a side-input is stable 1. Notice that sensitized and insensitized are not all inclusive in that there may be an input to a gate which may be neither if not enough side-inputs are stable. We defer the description of the algorithm to determine stability until Section 7.3.

7.2.3 Finding ODC's

Given the new notions of sensitized and insensitized from the previous subsection, Algorithm 7.1, similar in form to that given in [16], finds ODCs for speed-independent circuits. Essentially, the algorithm declares that a state is in a signal's ODC set if, in that state, the signal cannot affect any of its fanouts or it affects exactly one fanout which also has that state in its ODC set. As in the combinational case [16], the signal may not affect more than one of its fanout. Otherwise, ODC's found for circuits with reconvergent fanout can result in violations of complex-gate equivalence. It is not difficult to show that using ODCs found by our algorithm does not violate complex-gate equivalence. In addition, our definition of ODC's omits additional states which lead to hazards such as the problematic state [111] in the example described above.

7.2.4 Using ODC's to perform logic transformations

To verify that a transformation preserves complex-gate equivalence, our procedure checks to see if the transformation affects only ODC states, i.e.,

$$\left[\textit{ext_eval}(u')(s) \neq \textit{ext_eval}(u)(s)\right] \;\Rightarrow\; s \in \textit{ODC}(u)$$

for all states $s \in \Phi$. The complexity of this check is $O(\Phi)$ whereas verifying complex-gate equivalence directly is $O(\Phi \cdot E)$.

Even using our restricted ODC sets, however, logic transformations must still be chosen with care to maintain correctness. For example, acknowledgement wire forks are functionally redundant connections—they do not affect the external evaluation of any circuit node in any state. Hence, removing the connection always preserves complex-gate equivalence regardless of the relevant ODC set. Unfortunately, removing such a connection leads to hazards.

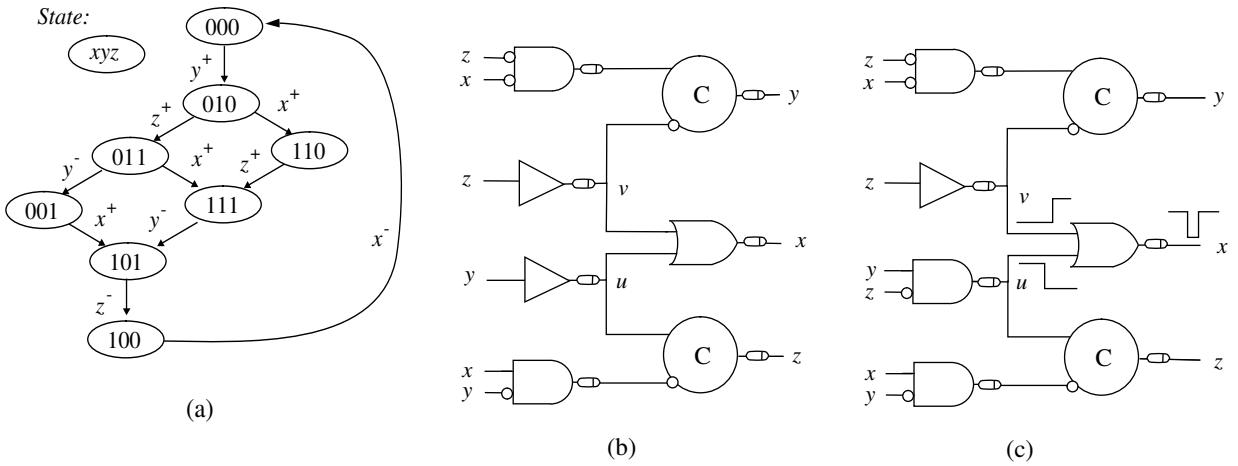


Figure 10: An example of where the combinational ODC algorithm leads to a hazard.

7.2.1 Representation

In the combinational circuit domain the most common representation of ODC's of a particular node u is a function which evaluates to *TRUE* when the value of u is not observable to any of the primary outputs of the circuit. In a synchronous sequential circuit this representation is complicated by signals separated by a clock cycle [16].

Fortunately, in an asynchronous circuit, the state of the circuit is not synchronized to a clock. Rather, the circuit state is held in the values of the primary input and output signals and stored in memory elements which can change at any time. Recall that this paper considers only externally-cut circuits, in which outputs of memory elements must be primary outputs and logic values of all primary outputs are given in each state. Hence, in each state, our circuits behave completely combinationally. Consider that the next state function of a two-input C-element with inputs a and b and output u is $f_u = ab + (a + b)u$. Since u is a primary output, the value of u in each state is known to be either 0 or 1. In states in which u is 0, the C-element acts like a 2-input AND gate. In states in which u is 1, the C-element acts like a 2-input OR gate. Therefore, in each state a combinational circuit representation can be obtained which behaves identically to the original circuit by changing all the memory elements to their combinationally equivalent gate in that state.

Now that we have converted our problem into a set of related combinational circuit problems, we can work from existing algorithms to determine ODC's. Informally speaking, a state is an ODC for an internal signal if no primary output is *sensitized*, defined below, to the internal signal in that state. As a result, we represent ODC's for node u as a set of SG states, referred to as $ODC(u)$.

7.2.2 Sensitized and insensitized signals

Our notion of sensitization is motivated both by the traditional combinational definition and hazard-freedom. In combinational circuits, a signal x is sensitized to a signal u if the Boolean difference function $\frac{\partial f_x}{\partial u}(s)$ evaluates to 1 [36]. In effect, Boolean difference $\frac{\partial f_x}{\partial u}(s)$ tells us what values we must assert on all other fanins of x, referred to as *side-inputs* [36], to make f_x sensitized to the value of u. Hence, in our notation if $f_x = OR(v, u)$, then $\frac{\partial f_x}{\partial u}(s) = \overline{ext_eval(s)(v)}$. It is important to realize that this sensitization is guaranteed only after all signals have settled. In an asynchronous circuit, however, in addition to the settled values, the transient behavior of the signals is relevant to avoid hazards. Therefore, the set $ODC(u)$ found using existing combinational algorithms can contain states which always lead to hazards in an asynchronous circuit.

Consider the example specification shown in Figure 10(a). When the corresponding correct circuit shown in Figure 10(b) is in states [011] and [111], both u and v evaluate to 1. Since in both states either u or v can enable the OR gate to rise, the combinationally-based algorithm given in [16] would conclude that they are ODC states of u. Using this set, standard algorithms would consider applying the logic transformation involving adding the connection z to the buffer associated with u, producing a 2-input AND gate, shown in Figure 10(c). While after applying this logic transformation the circuit remains complex-gate equivalent to the specification, it has a hazard.

be outputs of other gates. In fact, when the set and reset networks consist of AND gates with more than 2-inputs, it is still possible to split the AND gate inputs while preserving correctness.

6.4 Inverter bubble shuffling

An additional optimization technique is the shuffling of inverter bubbles [11]. Using DeMorgan's law, inverters can be pushed through gates, converting AND and OR gates to their duals. This is beneficial for two reasons. First NOR and NAND gates are smaller and faster than AND and OR gates in CMOS technology. Second, by applying this transformation, inverter bubbles on both ends of a wire can cancel out. Our CAD tool uses a simple greedy algorithm that finds a local optimum of inverter bubble placement.

7 Performing logic transformations and optimizations

Sharing of identical gates, signal network simplification, and bubble shuffling are guaranteed to be correctness-preserving. The implementation of these techniques, therefore, is straight-forward. The other two optimization techniques, removing gate inputs and gate symmetrization, however, must be performed selectively to ensure hazards are not introduced. Similarly, adding acknowledgement wire forks must also be done selectively.

In fact, the impact of such logic transformations can be far from obvious and naive analysis techniques are computationally prohibiting. This section describes an algorithm that efficiently verifies that a logic transformation does not introduce new hazards into the circuit, i.e., is *non-hazard-increasing*.

7.1 Overview

The key difference between legal logic transformations in synchronous logic and non-hazard-increasing transformations in speed-independent circuits is that in speed-independent circuits transient behavior of the logic is critical. As a result, transformations in speed-independent circuits are more constrained than logic transformations in synchronous circuits. While non-hazard-increasing transformations have been well studied for fundamental-mode circuits [48], these results do not generally apply to speed-independent circuits because of the additional concurrency allowed in speed-independent circuits between the circuit and its environment.

One approach to testing whether a logic transformation is non-hazard-increasing is to perform the transformation and verify that the new circuit has no additional hazards. While this approach is computationally infeasible with traditional verification techniques, it is possible with the conservative verification algorithm we developed in a related paper [3]. The complexity of this conservative verification algorithm is linear with respect to the size of the circuit and its SG for the class of circuits addressed in this paper.

Complete verification, however, is often not necessary. Since a logic transformation represents a local change in a circuit, only the affected parts of the circuit need to be verified. We describe how to take advantage of this fact using incremental verification, thereby providing an efficient means of checking whether logic transformations are non-hazard-increasing.

This section describes two techniques used in the incremental verification which target the two parts of our definition of correctness: complex-gate equivalence and hazard-freedom. The first technique manages observability don't care (ODC) information which is used to efficiently check that complex-gate equivalence is preserved while also filtering out additional transformations which are guaranteed to fail our conservative hazard-freedom check. The second technique manages stability information which is used to efficiently check that no hazards are introduced into the circuit.

7.2 Incrementally verifying complex-gate equivalence with ODCs

In traditional synchronous circuits, observability don't cares are used to characterize all legal transformations that are local to a gate in a circuit. This subsection explores the representation of ODCs in speed-independent circuits and explains how ODCs can be used to facilitate a local check for the preservation of complex-gate equivalence while also filtering out other hazard-increasing transformations.

6 Logic optimizations

This section describes a variety of logic optimization techniques that can improve the initial gate-level circuit. Namely, this section describes removing redundant gate inputs, sharing and merging gates, signal-network simplification, and inverter bubble shuffling. Algorithms used to implement the optimizations are discussed in Section 7.

6.1 Removing redundant gate inputs

A redundant gate input is one which can be removed from the circuit while preserving correctness. The removal of the input must preserve both complex-gate equivalence and hazard-freedom. A gate input can be functionally redundant, i.e., its removal preserves complex gate equivalence, while still not being a redundant gate input (e.g., an acknowledgement wire fork).

Redundant gate inputs arise for three reasons. First, since both set and reset functions of a signal are implemented separately and merged, they may duplicate needed logic. Sometimes the functions can be simplified while preserving correctness as illustrated in Section 7.2. Second, when an acknowledgement wire fork is added to a gate, other existing inputs of the gate may become redundant. Third, gate inputs may become redundant as a consequence of other optimizations, such as merging gates.

6.2 Sharing and merging gates

Two *internal gates*, gates whose outputs are internal signals, that have the same function can be safely shared as long as one of the internal gates is not used to acknowledge a request of an internal signal that is not also acknowledged by the other gate. Similarly, two internal gates that have exactly the opposite function can be implemented with one gate by adding inverter bubbles to the fanout of one of the gates. If two *output gates*, gates whose outputs are primary outputs, have the same function, then, instead of inverter bubbles, an inverter is necessary. Moreover, an internal and output gate can be shared, but the output gate must be replaced with a buffer or inverter, depending on if the two gates have the same or opposite functions, respectively. Otherwise, two outputs which are specified to be ordered may become concurrent in the implementation, violating the specification.

A generalization to the sharing gate optimization is when two arbitrary gates can be implemented with one gate. It is easy to see that within a sum-of-product, two AND gates that differ in only the phase of one literal can be safely merged into one gate. In addition, other AND gates in sum-of-product blocks that are logically distance-one apart can often be symmetrized and then merged. Similarly, if AND gates in different parts of the circuit can be safely symmetrized to the point that they have the same functionality, the AND gates can be shared.

Two different approaches have been proposed to implement this optimization. The first approach, suggested by Kondratyev et al. [25], is a one-step process, but is limited to unoptimized AND gates implementing region functions. They suggest to formulate the covering constraints for a group of excitation regions to see if a single cube cover exists for the group. As long as each excitation region is covered by exactly one cube, correctness is guaranteed. The second approach is a two-step process. A minimum set of signals is added to the two AND gates that make them distance-one apart yet preserve the correctness of the circuit. Then, the distance-one-apart gates are merged. Checking that the first step preserves correctness can be a computationally expensive task and its implementation is deferred to Section 7. The second step, merging distance-one-apart gates in a sum-of-products, however, is always correctness-preserving. Once merged, inputs to the gate may become redundant which can be removed by subsequent optimizations.

6.3 Signal network simplification

If the covers of either the set (reset) excitation regions of a signal u include their entire associated quiescent regions, then the set (reset) network is functionally equivalent to u. Unfortunately, even though the reset (set) network is not needed to implement signal u, it may still be necessary for acknowledgement of other internal signals. If all the acknowledgement wire forks attached to the reset (set) network can be relocated, the redundant logic can be removed. Specifically, u's reset (set) network and the C-element can be removed, reducing u's signal network to u's set (reset) network.

In addition, if the set and reset region networks of a signal network differ in phase the signal network can often be simplified. Consider the simplest case in which $S_u = v_1 v_2$ and $R_u = \overline{v_1}\ \overline{v_2}$. In this case, the two AND gates can be removed and the inputs to the C-element are reassigned to $S'_u = v_1$ and $R'_u = v_2$. Note that v_1 and v_2 may

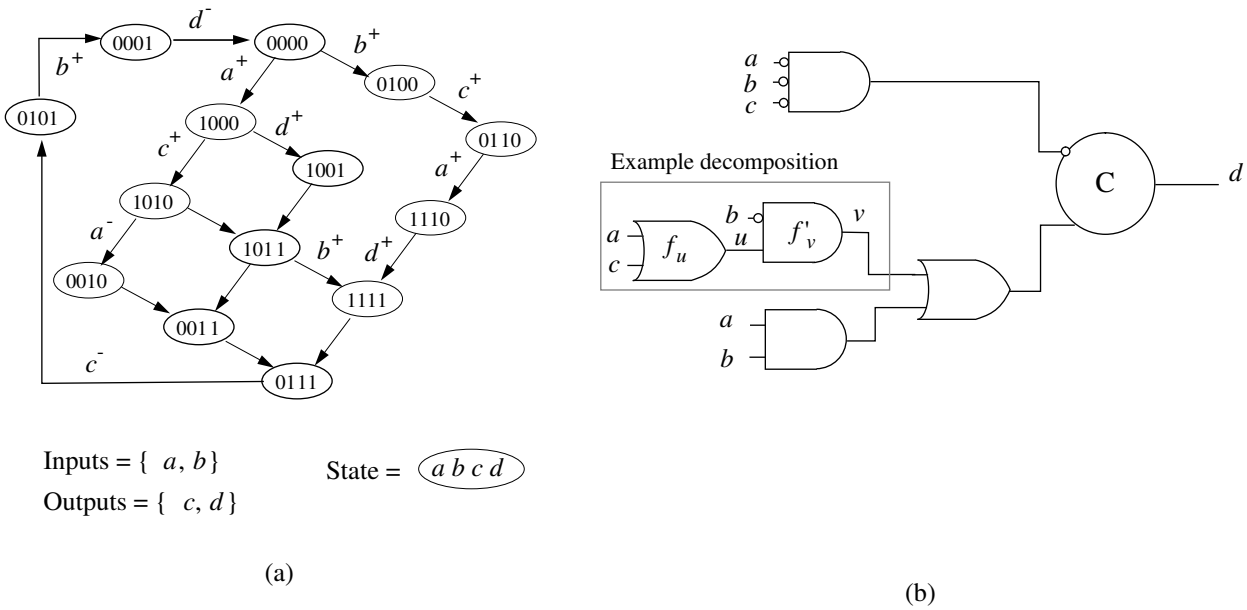


Figure 8: An example of decomposing complex gates.

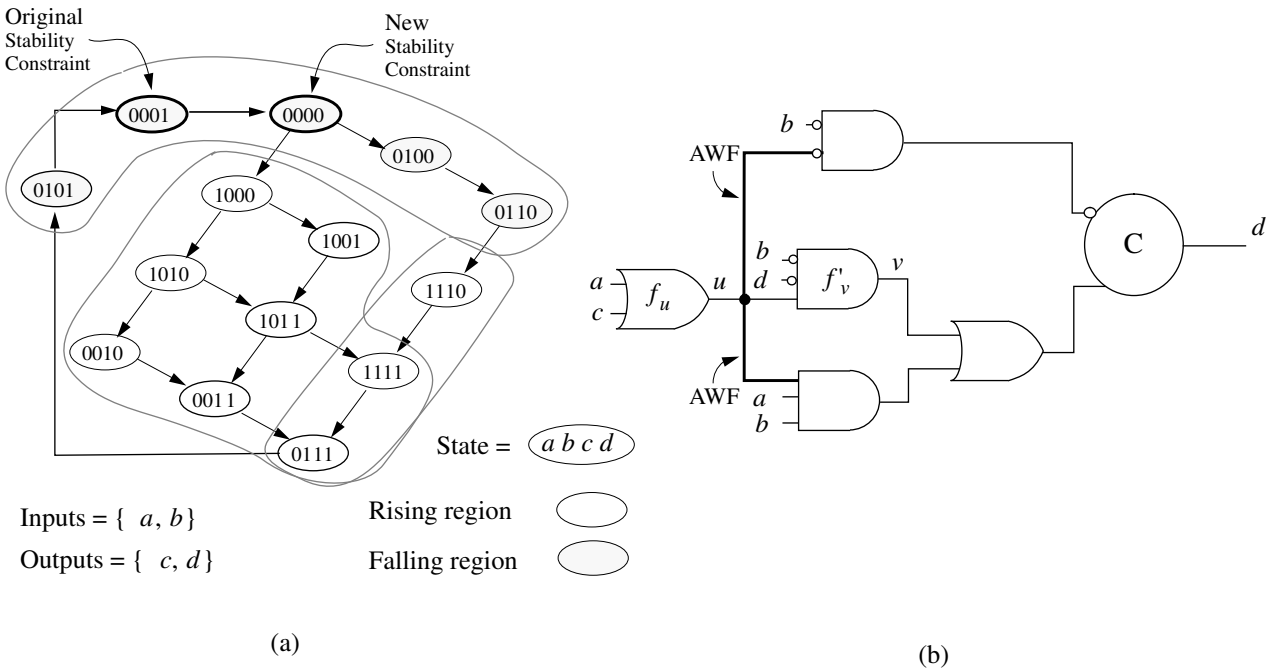


Figure 9: An example of acknowledging subregions.

new high-fanin gates, which must be subsequently decomposed. When this approach successfully decomposes all gates, the circuits generated are usually very efficient. The approach, however, is not guaranteed to successfully decompose all circuits. For such circuits, carefully designed complex gates can be used to ensure hazard-freedom, or the circuits' specifications can be altered by adding state variables. Identifying the class of specifications with choice for which all functions can be sufficiently decomposed while preserving the circuit's interface behavior is an open research problem. Nevertheless, our algorithm has successfully synthesized all SIS benchmark specifications with gates of fanin 4 or less.

Algorithm 5.1 (Repair decomposition)

```
/* Let G contain a new internal signal u with associated function f_u and a different function f'_v for signal v such
that f_v = f'_v ∘ f_u. */
repair_decomposition(u, v, G, M) {
  R = find_regions(u, G);
  R' = decompose_regions(R);
  foreach R'_i ∈ R'
    P_i = find_stability_constraints(v, R');
    if (satisfies_stability(R'_i, P_i) ≠ TRUE)
      c = find_cuts(R'_i, P_i);
      if (c == NULL)
        relax_constraints(u, v, G, M)
        c = find_cuts(R'_i, P_i);
        if (c == NULL)
          exit with failure;
      e = find_best_awf(R'_i, P_i, c);
      if (e == NULL)
        P_i = add_conn(u, v, G, M);
        e = find_best_awf(R'_i, P_i, c);
        if (e == NULL)
          exit with failure;
      build_awf(e);
}
```

Figure 7: Algorithm for repairing a decomposition.

Otherwise, the stability constraints limit the choices of acknowledgement wire forks of a signal u. They indicate that u must be stable before entering the stability constraint state. The function *find_cuts*() finds the output signal transitions that precede the stability constraint which might acknowledge u. For the falling subregion of the signal u in our example, there are no output signal transitions that cut the subregion which also precede the stability constraints. Hence, no acknowledgement wire forks can possibly be found that would guarantee that u is stable before state [0001] is entered. Fortunately, by adding additional inputs using the function *relax_constraints*() to the gate controlling v, the algorithm can sometimes relax the stability constraints. In this example, by adding $\overline{d}$ to the AND gate controlling v, the stability constraint is moved to state [0000]. Once moved, the function *find_cuts*() determines that a d^- transition can now acknowledge the falling subregion of u.

Once the set of acknowledging transitions is found, the function *find_best_awf*() locates the best potential acknowledgement wire forks. The function searches wire forks to all gates that control each potential acknowledging transition. A wire fork is a *potential acknowledgement wire fork* if the connection acknowledges the region and does not introduce additional acknowledgement or monotonicity violations. The implementation of checking the last requirement is addressed in Section 6. We note that one reason a wire fork is not a potential acknowledgement wire fork is that the wire fork would introduce extra unacknowledged requests on the acknowledging gate. By adding inputs to these acknowledging gates, using the function *add_conn*(), the extra requests can be prevented. Of course, since our goal is to minimize fanin and complexity, the addition of gate inputs is only done when necessary.

Finally, the function *find_best_awf*() returns the acknowledgement wire fork to the acknowledging gate with the lowest fanin, thereby attempting to avoid the creation of new high-fanin gates. In our example, one of the three subregions is already acknowledged and satisfies its stability constraints and requires no acknowledgement wire forks. The other two subregions are acknowledged by two acknowledgement wire forks shown in Figure 9(b). Once acknowledgement wire forks are built, inputs to the acknowledging gates may become redundant and may be removed with optimizations discussed in Section 6. In this example, both the inputs $\overline{a}$ and $\overline{c}$ of the top AND gate can be removed. As a result the net overhead for the acknowledgement wire forks is negligible—one additional connection.

Although successful for this example, this heuristic break-and-repair methodology can fail when *find_best_awf*() fails to find a connection that acknowledges the region and satisfies the stability constraints. When using this decomposition technique to break up high-fanin gates, the addition of acknowledgement wire forks can also introduce

connections to the circuit to reestablish hazard-freedom. The steps are iterated until all functions are of manageable size or no more functions can be successfully decomposed.

The most common function to be decomposed is a conjunction of many literals corresponding to a single-cube cover that covers relatively few states. For such functions, the obvious decomposition is a cascade of two smaller conjunctions. Since there may be many such decompositions, heuristics are used to order the attempts in such a way as to help find a decomposition leading to a correct circuit quickly.

For multi-cube covers there are many possible strategies for decomposition, including kernel extraction and function decomposition [17]. In [5], we describe an efficient decomposition heuristic in which the function is recursively decomposed into multi-level logic, guided by the relative ordering of signal transitions. We refer the reader to [5] for more details and focus in this paper on the repair aspects of our methodology which are general enough to be applicable to all decomposition techniques.

5.1 Acknowledgement wire forks

When a function is decomposed into two functions a new gate is introduced into the circuit and with it an associated arbitrary delay. While some decompositions maintain correctness [46], many introduce hazards into the circuit. When the only decompositions found introduce hazards, the algorithm attempts to repair the damage by adding connections between gates, called *acknowledgement wire forks*.

Decompositions can generate two sources of acknowledgement and monotonicity violations. Consider a decomposition of a function f_v into two functions f_u and f'_v such that $f_v = f'_v \circ f_u$. Recall from Section 3.2 that the new signal u has its own set of requests for changes that must be acknowledged and be monotonic for the circuit to be hazard-free. The requests for signal v are unchanged. Signal v, however, may now have new violations of monotonicity which must be removed. The monotonicity violations are caused by the added delay element in the network. Before the decomposition, the stability of some fanins of v may have made it insensitive to inputs that might cause hazards. After the decomposition, the functionality of these inputs may have been pulled out into f_u which may not stabilize soon enough to prevent the monotonicity violation on v.

5.2 Algorithm for repairing a decomposition

Our algorithm to repair a decomposition, depicted in Figure 7, is described using a circuit synthesized from a specification borrowed from [25], shown in Figure 8(a). In the synthesized circuit, shown in Figure 8(b), we attempt to decompose the complex gate $f_v = a\overline{b} + c\overline{b}$ into two functions $f_u = a + c$ and $f'_v = \overline{b}f_u$. The goal of the repair decomposition algorithm is to repair any acknowledgement or monotonicity violations caused by the presence of the new internal signal u. In other words, the algorithm attempts to ensure that all requests on u are acknowledged and monotonic. Recall from Section 3.2 that this means that every path through every signal region of u must be acknowledged. By adding an acknowledgement wire fork from u to some gate that controls an output signal, a transition of the output signal can be made to acknowledge previously unacknowledged paths. An obvious goal is to add a minimal number of such acknowledgement wire forks that guarantee that every path through the request is acknowledged. In other words, an optimal algorithm would find a minimal cut-set of acknowledging transitions.

Since finding the minimal cut-set is intractable, our algorithm uses a heuristic which may require more than the minimal number of acknowledgement wire forks, but works well in practice. The heuristic is to decompose each signal region into overlapping *subregions* that are independently acknowledged. The function *decompose_regions* finds the set of subregions for the region. Each subregion is formed from a different possible sequence of input choices made before and during the signal region. The distinguishing feature of a subregion is that both signal transitions between subregion states and signal transitions between a subregion state and a different signal region cut the subregion. As illustrated in Figure 9(a), the rising region of u in our example is decomposed into two overlapping subregions. One consisting of 8 states and the other of 3 states. These subregions arise from different ways that the rising signal region can be entered due to the input choice in the specification. The falling region of u is a single subregion.

Before examining possible acknowledgement wire forks, however, the algorithm must also consider the monotonicity violations of v caused by the decomposition. The function *find_stability_constraints*() identifies the *stability constraints* of the subregion. A stability constraint of a subregion is any state in which f_u must be stable in order to prevent a monotonicity violation of f'_v. In the example, a stability constraint of the falling region is state [0001] since u must be stable to avoid a monotonicity violation of v. That is, u must be guaranteed not to fall before b falls, since this can cause a glitch at v. If the stability constraint is initially satisfied, i.e. the function *satisfies_stability*() returns **TRUE**, no acknowledgement wire fork is needed for the current subregion.

The next step in the table construction is to find all state transitions which conflict with the entrance constraint in the initial cover or may conflict due to a context signal choice. For any state transition $s \xrightarrow{v} s'$, this is possible when s is not in the excitation region (i.e., $s \notin EC(u,k)$), s' is in the quiescent region (i.e., $s' \in QR(u,k)$), s' is in the initial cover (i.e., $s' \in TC(u,k)$), and v excludes s (i.e., $EC(u,k)(v) = \overline{s(v)}$). For each entrance conflict detected, the algorithm adds a new column to the table again with a cross in each row corresponding to a context signal that would exclude the conflicting state. If the signal v in the state transition is a context signal, the state s' only needs to be excluded if v is included in the cover. This implication is represented with a dot being placed in the row corresponding to the signal v.

In a single pass through the state graph, all the CC tables can be constructed. Returning to our example, the CC table for the excitation region $ER(c,1)$ is given in Figure 3. For this excitation region, the enabled state is [0100]

CC Table				
	Covering Section		Closure Section	
	[1100]	[1101]	[1110]$\xrightarrow{a}$[0110]	[1111]$\xrightarrow{d}$[1110]
a	×	×	○	×
c			×	×
d		×		○

Table 3: The CC table for the single-cube covering algorithm applied to $ER(c,1)$.

and b is its only trigger signal. The covering section includes states [1100] and [1101] because all other states are either in the excitation or quiescent region or are excluded by the trigger signal b. There are two closure columns. The first, corresponding to the transition [1110]$\xrightarrow{a}$[0110], indicates that if a is included then state [0110] must be excluded. The only context signal that excludes this state is c. The second closure column corresponds to the transition [1111]$\xrightarrow{d}$[1110] and is formed similarly. Note that the transition [1101] $\xrightarrow{c}$[1111] does not have a column since c does not exclude state [1101].

When the construction of the CC table is successful, the table is solved using essentially the same reduction algorithm used in the general case outlined above. In this case, however, ties that occur in Rule 4 are resolved by picking the rule that provides symmetry between different regions of the same signal. This symmetry can often be exploited later during logic optimizations (see Section 6). Returning to our example, the table is solved as follows. First, row a is picked since it is an essential row (Rule 1), removing it as well as columns [1100], [1101], and [1111]$\xrightarrow{d}$[1110] from the table. Since this removes a dot in column [1110]$\xrightarrow{a}$[0110], this column is covered next. To accomplish this, row c is picked since it is an essential row (Rule 1), removing the column [1110]$\xrightarrow{a}$[0110] solving the table. The resulting correct cover, as expected, is the same single cube $\overline{a}b\overline{c}$ found by the general algorithm.

When a trigger signal is not persistent or when the CC table construction fails, we can use the more general algorithm described in Section 4.2.1 to find a multi-cube cover. Alternatively, we can change the specification by constraining concurrency [37] or by adding state variables [25, 52, 6] such that a single-cube cover can be found. We note that this alternative may not be possible *without changing the interface behavior of the circuit* (i.e., without constraining an input signal).

5 Implementing region functions

Once an optimal cover is found, the cover must be implemented using limited-fanin basic gates. Often, the cover is reduced to a single cube with only a small number of literals where an AND gate that implements the cube is usually found in the target library. In addition, simple multi-cube covers can often be implemented safely with carefully designed and-or-invert gates. For larger, more concurrent specifications, however, single-cube covers that correspond to high-fanin AND gates or multi-cube covers may be needed. Therefore, it is important to consider how such covers can be implemented hazard-freely at the gate-level.

Our approach is to iteratively decompose the function into smaller functions until all base functions can be implemented by gates in the target library. This decomposition follows a break-and-repair strategy. First, the function is decomposed into two functions introducing a new internal signal into the circuit, adding a new delay element into the circuit which often causes a hazard, i.e., breaks the circuit. The repair step of our strategy adds

u,k	*set/reset*	*EC*(u,k)	*TC*(u,k)
$c,1$	*set*	0100	X1XX
$c,2$	*set*	1101	XXX1
$c,3$	*reset*	0010	X0XX
$d,1$	*set*	1100	X1XX
$d,2$	*reset*	1111	XX1X

Table 2: Enabled cubes and trigger cubes for our example, where the cube vector is $\langle a,b,c,d\rangle$

than the one in [12]). Otherwise, the single-cube cover would not cover all excitation region states. When a single-cube cover exists, an excitation region *ER*(u,k) can be sufficiently approximated using a cube called an *enabled cube*, denoted *EC*(u,k), defined on each signal v as follows:

$$\textit{EC}(u,k)(v) \equiv \begin{cases} 0 & \text{if } \forall s \in \textit{ER}(u,k)\ [s(v)=0] \\ 1 & \text{if } \forall s \in \textit{ER}(u,k)\ [s(v)=1] \\ X & \text{otherwise} \end{cases}$$

If a signal is 0 or 1 in the enabled cube, it can be used in the cube implementing the region. A cube, such as the enabled cube, implicitly represents a set of states in the obvious way. The set of states implicitly represented by the enabled cube is always a superset of the set of excitation region states.

Each single-cube cover in the implementation is composed of *trigger signals* and *context signals*. For a given excitation region, a trigger signal is a signal whose firing can cause the circuit to enter the excitation region while any non-trigger signal which is stable in the excitation region can be a context signal. The set of trigger signals for an excitation region *ER*(u,k) can also be represented with a cube called a *trigger cube*, denoted *TC*(u,k), defined as follows for each signal v:

$$\textit{TC}(u,k)(v) \equiv \begin{cases} s'(v) & \text{If } \exists s,s' \left[(s \xrightarrow{v} s') \wedge (s \notin \textit{ER}(u,k)) \wedge (s' \in \textit{ER}(u,k))\right] \\ X & \text{otherwise} \end{cases}$$

It is easy to show that in order for a single-cube cover to satisfy the covering constraint it must contain all its trigger signals. Since only persistent signals can be included in a single-cube cover, a necessary condition for a single-cube cover to exist is that all trigger signals be persistent. In other words, for a given excitation region *ER*(u,k), the following must be true:

$$\forall v \in A_{\textit{Spec}} \left[(\textit{TC}(u,k)(v) = X) \vee (\textit{TC}(u,k)(v) = \textit{EC}(u,k)(v))\right]$$

The enabled cubes and trigger cubes are easily found with a single pass through the state graph. The enabled cubes and trigger cubes corresponding to all the excitation regions in our example are shown in Table 4.2.2. Notice that every trigger signal is persistent and our algorithm proceeds to find the optimal single-cube cover.

Our algorithm starts with a cube consisting only of the trigger signals. If this cover contains no *conflicts*, i.e., states that violate either the covering or entrance constraint, we are done. This, however, is often not the case, and context signals must be added to the cube to remove any conflicting states. For each conflict detected, the procedure determines the choices of context signals which would exclude the conflicting state. Finding the smallest set of context signals to resolve all conflicts is a covering problem. Due to the implication in the entrance constraint, inclusion of certain context signals may introduce additional conflicts which must be resolved. Therefore, the covering problem is *binate*.

To solve this binate covering problem, we again create a CC table for each region. There is a row in the CC table for each context signal, and there is a column for each conflict and each conflict that could potentially arise from a context rule choice. An entry in the table contains a cross ($\times$) if the context signal resolves the conflict. An entry in the table contains a dot ($\circ$) if the inclusion of the context signal would require the conflict to be resolved.

To construct the table for a given excitation region *ER*(u,k), the algorithm first finds all states in the initial cover which conflict with the covering constraint. In other words, a conflict exists if a state s is (implicitly) contained by *TC*(u,k) but is not in the excitation or associated quiescent region. If a conflict exists, the algorithm adds a new column to the table with a cross in each row corresponding to a context signal v that would exclude the conflicting state (i.e., *EC*$(u,k)(v) = \overline{s(v)}$).

CC Table					
	Covering Section	Closure Section			
	[0100]	[1110]$\xrightarrow{a}$[0110]	[1101]$\xrightarrow{c}$[1111]	[1101]$\xrightarrow{c}$[1111]	[1101]$\xrightarrow{c}$[1111]
$\overline{a}b$	×	o			
$\overline{a}b\overline{c}$	×				
ac		×	o		
$a\overline{b}c$					
bc		×		o	
$\overline{a}d$					
$\overline{b}d$					
cd					o

Table 1: The CC table for the general covering algorithm applied to $ER(c, 1)$.

Rule 3: (Remove dominating columns) A column C_j dominates a column C_i if it has all the crosses and dots of C_i. If C_j dominates C_i, then C_j is deleted.

Rule 4: (Remove dominated rows) A row R_i dominates a row R_j if it (a) has all the crosses of R_j, and (b) for every column C_p in which R_i has a dot, either R_j has a dot in C_p or there exists a column C_q in which R_j has a dot, such that, disregarding the entries in rows R_i and R_j, C_p dominates C_q. If R_i dominates R_j, then R_j is deleted together with all columns in which it has dots.

Rule 5: (Remove rows with only dots) If a row only has dots, then the row is deleted together with all columns in which it has dots.

It is important to note that when applying rule 4, two rows may mutually dominate each other. To break this tie, our algorithm removes the row corresponding to the implicant composed of the larger number of literals.

The table is completely solved when all columns are eliminated, and the resulting cover is the set of essential rows selected by Rule 1. In our limited experience, these reduction rules are usually sufficient to solve the table. For some cases, however, the reduction rules do not reduce the table completely, leaving a *cyclic table*. To solve the cyclic table, we currently use traditional branching techniques [35] and are exploring the use of newer branch and bound techniques [9, 23, 44].

The reduction steps solve the table depicted in Table 1 as follows. First, the rows ac, bc, and cd along with the three columns labeled [1101]$\xrightarrow{c}$[1111] can be removed by Rule 2. Moreover, rows $\overline{a}b$, $a\overline{b}c$, $\overline{a}d$, and $\overline{b}d$ are dominated by row $\overline{a}b\overline{c}$ and can be removed along with the column [1110]$\xrightarrow{a}$[0110] by Rule 4. The remaining candidate implicant, $\overline{a}b\overline{c}$ is essential and is picked by Rule 1, solving the table. Note that in this case the table can only be solved by selecting an implicant that is not prime.

This example motivates two optimizations. First, prime implicants that cover only don't care states need not be considered in the generation of the candidate implicants, since such candidate implicants are never part of an optimal cover. Second, quiescent region states that are not reachable from some state in the excitation region through a path consisting of only other quiescent region states can be put in the off-set when determining the prime implicants, since the entrance constraint excludes them from being part of a correct cover. These optimizations can make the initial CC table significantly smaller.

4.2.2 Single-cube algorithm

The above binate covering formulation is often more general than needed since many region functions can be implemented with a single-cube cover. In this subsection, we present a more efficient algorithm designed to find an optimal single-cube cover if one exists. This algorithm is derived from an algorithm used to synthesize complex-gate timed circuits [40] by adding the necessary closure constraints needed to handle gate-level hazards, and it has also been applied to the synthesis of gate-level timed circuits [41]. Our synthesis procedure initially attempts to find single-cube covers, and only when this algorithm fails is the general algorithm described above invoked.

For a single-cube cover to hazard-freely implement a region function, all literals in the cube must correspond to signals that are *persistent*, i.e., constant throughout the excitation region (this is a slightly more general definition

and *FALSE* otherwise. A conjunctive function over these variables is composed of two types of disjunctive clauses. This function is *TRUE* when the included cubes make up a correct cover.

First, a *covering clause* is included for each state s in the excitation region of an output signal. Each clause consists of a disjunction of candidate implicants that cover s, i.e.,

$$\bigvee_{i:s \in c_i} l_i.$$

To satisfy the covering clause for a state s, at least one l_i must be set to *TRUE*. This means that one cube that covers s must be included in the cover. It follows that the set of covering clauses for an excitation region guarantee all excitation region states are covered. Since all candidate implicants are guaranteed not to include states outside of the excitation and associated quiescent region, the cover is guaranteed to satisfy the covering constraint.

Second, for each candidate implicant c_i, a *closure clause* is included for each of its implied states $s \in IS(c_i)$. Each closure clause represents an implication that states if the Boolean variable associated with the cube c_i is true then the implied state s must be covered. To fit into a conjunctive form the implication is translated to the equivalent disjunction, i.e.,

$$\overline{l_i} \vee \bigvee_{j:s \in c_j} l_j.$$

A closure clause states that if c_i is in the cover, some other cube must also be selected that covers the implied state s. The covering and closure clauses together ensure that the cover satisfies the entrance constraint.

When the entire conjunctive function is satisfied, the corresponding cover is correct. Our goal is to find an assignment of Boolean variables that satisfies the function with the minimum cost. The cost function we minimize is the number of implicants, though, the number of literals can also be used. Since the implication introduces negated variables into the satisfiability product-of-sums framework, our optimization problem is a binate covering problem.

We now present an algorithm to find a cover using the minimum number of candidate implicants. First, the algorithm finds the prime implicants for each region function. Second, it uses this set to find all the candidate implicants. Then, it solves the binate covering problem represented as a *covering and closure table* (or *CC table*) [21], using traditional reduction and branching techniques. We are currently exploring newer algorithms that may solve our binate covering problems more efficiently [9, 23, 29, 44].

In order to find the set of prime implicants, our algorithm partitions the Boolean space into three sets, the on-set, the off-set, and the don't care-set. The on-set is composed of every state in the excitation region. The don't-care set is composed of every state in the associated quiescent region as well as every unreachable state. The off-set is composed of every other reachable state. The prime implicants are found using standard techniques [8].

Next, the algorithm expands the set of prime implicants to include all candidate implicants using the procedure described in [21]. The algorithm first seeds the list of candidate implicants with the prime implicants, sorted by the number of literals in the implicant. Beginning with the candidate prime with the fewest number of literals, the algorithm considers all implicants extended with a literal not already used in the prime. If any new implicant satisfies the conditions given above then the algorithm inserts it into the list. Each subsequent implicant is considered in order until no new candidate implicants can be added.

To solve the binate covering problem, a CC table is constructed to represent the conjunctive function described above. The table has one row for each candidate implicant and one column for each clause. The columns are divided into a *covering section* and a *closure section*, corresponding to covering and closure clauses. In the covering section, for each excitation region state s, a column exists containing a cross in every row corresponding to a candidate implicant that covers s. In the closure section, for each implied state s of each candidate implicant c_i, a column exists containing a dot in the row corresponding to c_i and a cross in each row corresponding to a candidate implicant c_j that covers the implied state s.

As an example, the CC table for excitation region $ER(c, 1)$ in our example is depicted in Table 1. The CC table is solved using the reduction rules described in [21], which are listed here for convenience:

Rule 1: (Select essential rows) If a column contains only a single cross and blanks elsewhere, then the row with the cross must be selected. This row is deleted together with all columns in which it has crosses.

Rule 2: (Remove columns with only dots) If a column has only a single dot and blanks elsewhere, the row containing the dot must be deleted together with all columns in which it has dots.

The relationship between cover constraints and the acknowledgement of the region function requests is subtle. Kondratyev et al. in [25] misinterpreted these conditions, which we first presented in [5], and erroneously claimed that our covering constraints do not guarantee acknowledgement and allow hazardous implementations to be synthesized [24]. In fact, the constraints described in their paper are not complete, allowing hazardous covers such as the one shown in Figure 6. This ambiguity, however, can be resolved with a simple amendment to their theory, making the two sets of covering constraints essentially equivalent [24].

4.2 Algorithms

While standard logic minimization techniques exist to find optimal covers [8], they do not guarantee hazard-free logic. In particular, they are not suited to solve our more constrained covering problem. To guarantee hazard-free logic, we included the notion of an entrance constraint which requires that a correct cover can be entered only through excitation region states. The entrance constraint ensures that if a state in the quiescent region is covered than each of its predecessor states must also be covered. This implication leads to a *binate covering problem* [22].

This section presents algorithms to solve this binate covering problem obtaining an optimal region function. In general, a region function is composed of a set of *cubes*. A cube is a set of *literals* which are either an external signal or its complement. First, we present a general algorithm that finds an implementation for each region function composed of the minimal number of cubes. It is often the case, however, that a region function can be implemented using only a single cube. If this is the case, we use a more efficient algorithm which finds a single-cube implementation for each region function composed of the minimal number of literals.

4.2.1 General algorithm

The goal of the general algorithm is to find an optimal sum-of-products function for each region function that satisfies our definition of a correct cover. The final hazard-free multi-level logic implementation of this function is obtained using the decomposition techniques described in the next section.

The sum-of-products cover consists of a disjunction of *implicants*. An implicant of an excitation region is a cube that may be part of a correct cover. In other words, a cube c is an implicant of an excitation region $ER(u,k)$ if the set of reachable states covered by c is a subset of the states in the union of the excitation region and associated quiescent region, i.e.,

$$[c \cap \Phi] \subseteq [ER(u,k) \cup QR(u,k)].$$

A *prime implicant* is an implicant which is not contained by any other implicant of the excitation region [35]. A sum-of-products cover is optimal if there exist no other cover with fewer implicants.

To capture the entrance constraint, each implicant c is said to have a corresponding set of *implied states* (denoted $IS(c)$). An implied state of a cube c is a state that is not covered by the implicant but due to the entrance constraint must be covered if the implicant is to be part of the cover. More precisely, a state s is an implied state of an implicant c for the excitation region $ER(u,k)$ if s is not covered by c and is a predecessor of a state that is both covered by c and in the quiescent region, i.e.,

$$IS(c) = \left\{ s \;\middle|\; s \notin c \wedge \exists s' \left[(s,s') \in \Gamma \right) \wedge (s' \in c) \wedge (s' \in QR(u,k)) \right] \}$$

It is important to note that an implicant may have implied states that are outside the excitation and quiescent regions and cannot be covered by any correct cover. For this reason, some instances of our covering problem cannot be solved using only prime implicants and non-prime implicants must be considered.

To this end, we define an implicant to be a *candidate implicant* if there exists no other implicant which properly contains it and has a subset of the implied states. In other words, c is a candidate implicant if there does not exist an implicant c' that satisfies the following two conditions:

$$\begin{aligned} c' &\supset c \\ IS(c') &\subseteq IS(c). \end{aligned}$$

Notice that prime implicants are always candidate implicants, but that a candidate implicant need not be prime. It is easy to prove that an optimal cover always exists that consists of only candidate implicants.

Using these candidate implicants, our covering problem is formulated by creating a binary function in conjunctive (product-of-sums) form to be satisfied with minimum cost. The binary function is defined over a set of Boolean variables l_i, one for each candidate implicant c_i. The variable l_i is *TRUE* if the cube c_i is included in the cover

$QR(u,k)$. For example, there are two excitation regions for the signal c to rise in Figure 1. The first, denoted $ER(c,1)$, is the set of states $\{[0100]\}$ and the second, denoted $ER(c,2)$, is the set of states $\{[1101]\}$. Both these excitation regions share the same quiescent region $QR(c,1) = QR(c,2) = \{[0110],[1110],[1111]\}$.

4.1.2 Correct covers

The *cover* of an excitation region $C(u,k)$ is a set of states in which the corresponding region function externally evaluates to one. A cover is a correct cover if it satisfies two conditions. First, it must satisfy the *covering constraint* which says that the reachable states in the cover must include the entire excitation region but must not include any states outside of the union of the excitation and associated quiescent region, i.e.,

$$ER(u,k) \subseteq [C(u,k)) \cap \Phi] \subseteq [ER(u,k) \cup QR(u,k)].$$

Second, it must satisfy the *entrance constraint* which says that a correct cover must only be entered through excitation region states, i.e.,

$$\left[s \notin C(u,k) \wedge s' \in C(u,k) \wedge (s,s') \in \Gamma\right] \Rightarrow s' \in ER(u,k).$$

As proven in Appendix E, ensuring that all region function covers are correct covers is sufficient to guarantee that the standard C-implementation is a correct circuit provided that each region function is implemented with a single atomic gate. The covering constraint guarantees that the circuit is complex-gate equivalent to the specification. A correct cover guarantees that each region function is only allowed to turn on when it is actively trying to fire u. This guarantees that every request of the region functions, the OR gates, and the C-element is acknowledged by the output signal changing. Moreover, it guarantees that no two inputs to the OR gates are simultaneously one, avoiding what has traditionally been called a delay hazard [1]. We also show that a correct cover guarantees the monotonicity of all requests on all gates in the signal network.

To illustrate the importance of the entrance constraint in correct covers consider the cover and corresponding standard C-implementation for the output signal c shown in Figure 6. The cover $\{[0100],[0110],[0101],[0111]\}$

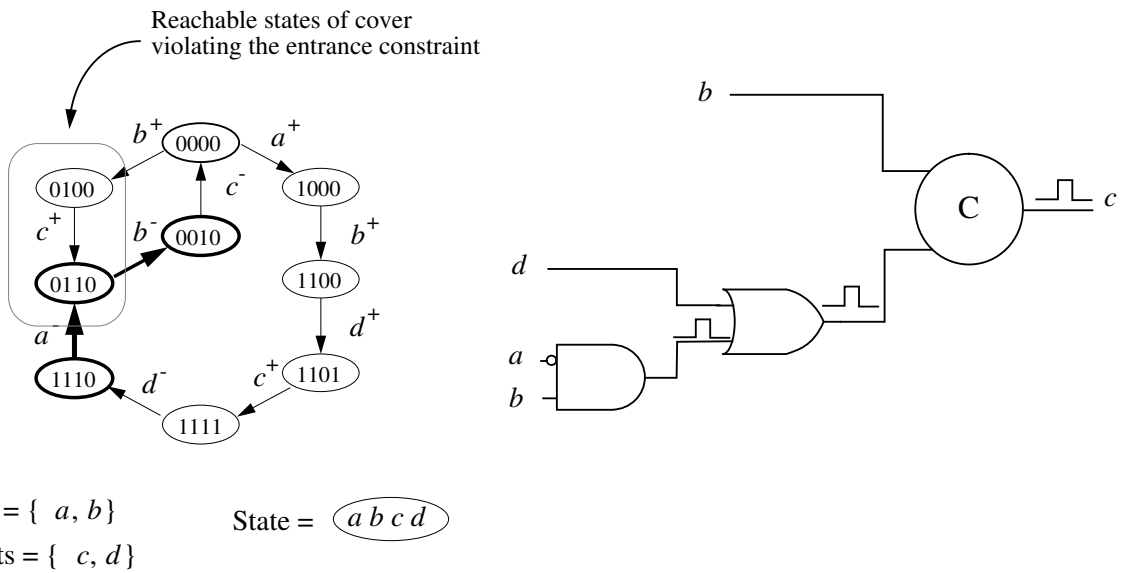


Figure 6: (a) A cover violating the entrance constraint for a set excitation region of the signal c, and (b) the corresponding hazardous logic implementation.

(which includes unreachable states [0101] and [0111]) fails to satisfy the entrance constraint since it can be entered from the state [1110] which is not in the excitation region. As a result, the corresponding region function *AND-N-1*(a,b) can turn on and off without being acknowledged. This can cause a glitch at the output of the AND gate which may cause the circuit to malfunction.

prove that our synthesis algorithm generates hazard-free circuits and that our logic transformations do not introduce hazards.

4 Standard C-implementation

The first step of our proposed procedure is to generate an optimal block-level implementation. To accomplish this, set and reset *excitation regions* of each output signal are identified in the SG and are implemented in independent blocks called *region functions*. Intuitively, each region function implements a different output signal *transition* in a higher-level specification. For example, the two c^+ transitions in the STG depicted in Figure 1(a) are independently implemented using two region functions. The region functions for each output are merged into a *signal network*, depicted in Figure 5, using two OR gates and one two-input Muller C-element, an asynchronous memory element. This C-element has a non-inverted input from the *set network* S_u, an inverted input from the *reset network* R_u, and one output u. The next state equation is $f_u = (S_u + \overline{R_u}) \cdot u + S_u \cdot \overline{R_u}$. In other words, when S_u is high and R_u is low, the signal u is driven high. When S_u is low and R_u is high, u is driven low. Otherwise, the signal u retains its old value. The collection of signal networks, one for each output, is referred to as the circuit's standard C-implementation. Note that while this implementation strategy may not immediately represent an optimal hazard-free circuit, particularly when logic can be shared across set and reset region functions, optimizations presented in Section 6 can be used to significantly improve circuit quality.

To ensure hazard-freedom of the block-level implementation, the covers of the region function must satisfy certain *correct cover* constraints. This section first develops these constraints and describes their relationship to ensuring hazard-freedom, i.e., acknowledgement and monotonicity. Next, this section casts the problem of finding the optimal correct cover into a binate covering problem and presents two efficient algorithms to find the optimal correct cover.

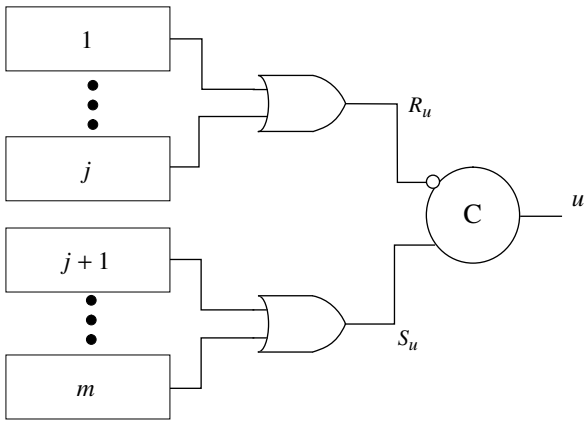


Figure 5: A signal network for an output signal u consisting of m region blocks denoted 1 through m.

4.1 Theory

4.1.1 Excitation and quiescent regions

For each output signal, the state graph is partitioned into a collection of excitation and quiescent regions. An excitation region is a maximally connected set of states in which the output signal is both enabled and at a constant value. Excitation regions are divided into two types depending on the value of the output signal in the excitation region. If the value is 0, the region is a *set excitation region* since in all excitation region states the output signal is enabled to rise; otherwise the region is a *reset excitation region*. A maximally connected set of states in which the output signal is not enabled is called a quiescent region of the output signal. In a determinate speed-independent SG, there exists at most one quiescent region directly reachable from a given excitation region, but a quiescent region may be entered from multiple excitation regions. The excitation regions for a signal u are indexed with the variable k and the k^{th} excitation region of signal u is denoted $ER(u,k)$. The associated quiescent region is denoted

slow, it may not rise before the circuit enters state [0010] and can later be disabled. This can cause the sequence of transitions u^-, c^-, v^+, c^+, v^-, c^-, representing a dynamic hazard at signal c.

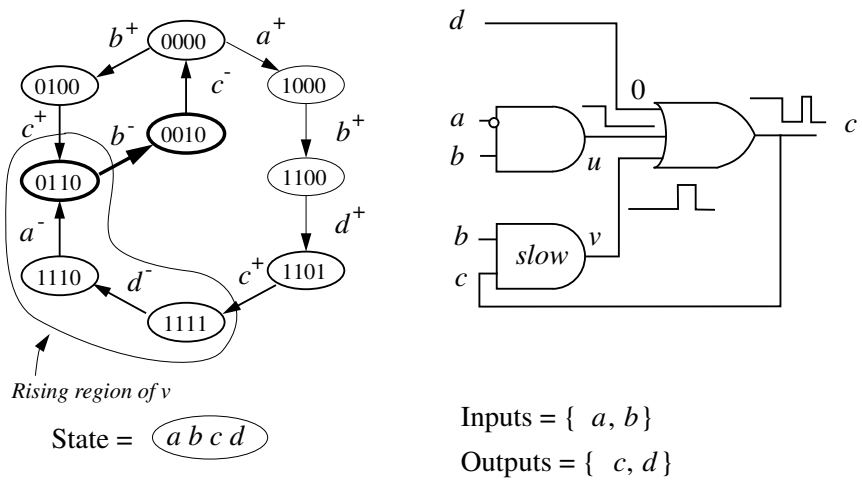


Figure 3: Illustration of a naive decomposition introducing an acknowledgement violation into the circuit.

As mentioned earlier, the request for a signal transition must be monotonic. Consider the rising request of c in Figure 4 corresponding to the signal region consisting of states $\{[0100], [0110], [1110], [1111], [1101]\}$. In state transition $[1111] \xrightarrow{d} [1110]$, the request must travel through the AND gate driving signal v before it reaches the OR gate that drives signal c. To ensure that c sees this request glitch free during this state transition, c must be *insensitive* to d, the falling input to the OR gate. In other words, the request must travel through the AND gate and drive v high before d can fall. Unfortunately this ordering is not guaranteed and the sequence of transitions d^-, c^-, v^+, c^+ depicted in Figure 4 can occur, representing a static 0-hazard on output c.

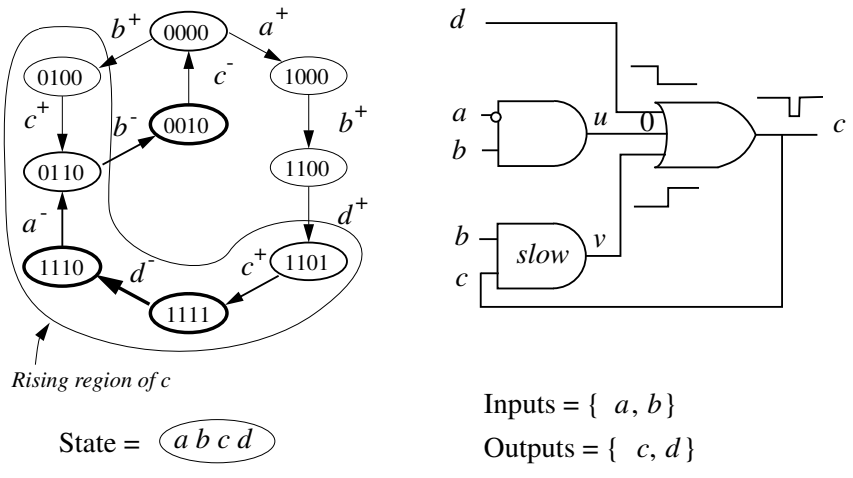


Figure 4: Illustration of a naive decomposition introducing monotonicity violation into a circuit.

Appendix C formalizes a procedure that conservatively identifies when internal signals are guaranteed to be high or low. Appendix D then formalizes the corresponding notions of acknowledgement and monotonicity and shows that they are sufficient (but not necessary) to guarantee hazard-freedom. We use these sufficient conditions to

gate equivalent to its specification. Informally speaking, a circuit is said to be hazard-free if logic glitches cannot occur at the output of any gate. If every output signal is implemented with a single complex gate, complex-gate equivalence is sufficient to guarantee that the circuit is hazard-free.

3.2 Gate-level implementations

Gate-level implementations are circuits in which each output signal is implemented with a netlist of basic gates. The key difference between gate-level implementations and complex-gate implementations is the presence of internal signals and their associated arbitrary delay elements whose behavior is not modeled in the SG. Since these arbitrary delays can result in unexpected changes in the arrival order of signal transitions, naive decompositions of complex gates into netlists of basic gates often introduce hazards. For example, in this section we show that the naive sum-of-products decomposition of the complex-gate implementation shown in Figure 2(c) has numerous sources of hazards.

To find hazard-free decompositions, we must understand the sources of such hazards and ensure that our decompositions do not introduce any of these sources. This subsection describes a classification of hazard sources based on an interpretation of the request/acknowledge handshaking protocol on circuit gates. First we describe our interpretation of this protocol in general terms and then use the above example to explain the details.

3.2.1 The request/acknowledge protocol

In its general form, the request/acknowledge handshaking protocol is a method of managing the communication between modules. For example, a request can represent that data is valid on the inputs of a function block F and that F should begin to compute. A corresponding acknowledgement from F indicates that the computation has been completed and that valid data exists on its outputs.

For gate-level speed-independent circuits, a request is a set of states in which a circuit signal u is supposed to change to a particular value. An acknowledgement of this request is any primary output transition that cannot occur until after the signal reaches the particular value. A necessary condition to guarantee hazard-freedom is that every request is *acknowledged* before a new request is sent to the same signal. Otherwise, the two requests can collide which causes a hazard.

Acknowledgement, however, is a necessary but not a sufficient condition for hazard-freedom. Since the request may need to travel through many gates until it reaches the input to the gate that drives the signal, the request must be guaranteed to be glitch-free. In other words, the request must reach the signal *monotonically*. Acknowledgement and monotonicity together form *necessary and sufficient conditions* for hazard-freedom. These concepts are formalized in Appendices A and B.

3.2.2 Acknowledgement and monotonicity

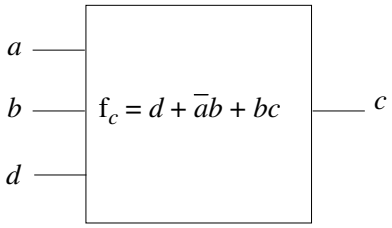
Since our circuits are externally-cut, only acyclic combinational logic connects internal signals to external signals. Hence, each circuit signal is either driven high or driven low in each state s but does not necessarily reach this value before the circuit leaves state s. The value that a signal u is driven to is referred to as the *external evaluation* of u in state s, denoted *ext_eval*$(s)(u)$. A *signal region* of u is a maximal *connected* set of states in which u has the same external evaluation, where a set of states is connected if any two members are mutually reachable through an undirected path of states in the set. Each signal region is a single request for the signal u to change to its external evaluation. For example, a request for signal v to rise is represented by the signal region $\{[1111], [1110], [0110]\}$ because v evaluates to 1 in all these states, as depicted in Figure 3.

A path of states through a signal region of u is acknowledged by a transition of a primary output v if the path includes a state transition $s \xrightarrow{v} s'$ which cannot fire until after u reaches its new external evaluation. A signal region is *acknowledged* if all paths through the region are acknowledged. There may be many primary output transitions that can acknowledge the same path through a request. In fact, in the presence of input choice, many primary output transitions may be needed to acknowledge a single request.

Every request must be acknowledged before a new request is sent to the same signal to prevent two requests from colliding. Consider the decomposition of the complex gate for the signal c described in Figure 3. Consider the highlighted path through the rising request for v following states [0110] and [0010]. Notice that the path is not acknowledged since it does not contain any output signal transition that acknowledges v rising. As a result, this rising request on v can collide with the subsequent falling request on v. In other words, if the AND gate bc is

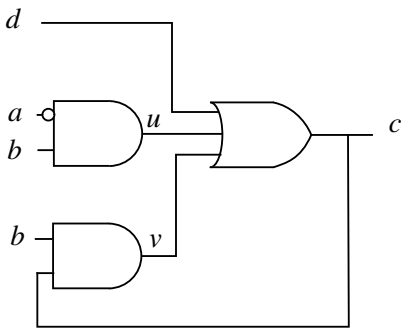
Next State Table	
State	**Value**
0000	0
0100	1
0110	1
0010	0
1110	1
1111	1
1101	1
1100	0
1000	0

(a)



Complex-gate implementation

(b)



Naive SOP implementation

(c)

Figure 2: (a) The next-state table for c, (b) a complex-gate implementation for the SG depicted in Figure 1, and (c) the naive sum-of-products implementation.

Figure 1(c) depicts a circuit implementation of the SG shown in Figure 1(a). Formally, the circuit is modeled by $\langle I, O, N, E, F \rangle$, where $I = \{a, b\}$, $O = \{c, d\}$, $N = \{e, g\}$. There are 10 edges in E including (a, d), (b, d), and (c, d) which correspond to the fanins of d. The functions $f \in F$ that are associated with each internal and output signal are $f_d =$ *AND-N-3*(a, b, c), $f_e =$ *AND-N-1-3*(a, b, c), $f_g =$ *OR*(d, e), and $f_c =$ *C-ELEMENT*(c, b, g). The circuit signals a and b are input signals and thus have no associated gate function; their behavior is derived from the specification. Notice that the inverter bubbles are included in the gate function. For example, the gate *AND-N-3* is an AND gate whose third input is inverted.

3 The gate-level synthesis problem

The gate-level synthesis problem is to find a netlist of library gates that implements a given state graph. Previously, Chu [12] and Meng [37] solved the problem by assuming the gate library had all possible complex gates such that each output function could be implemented with a single complex gate. When the gate library is more constrained, such as is in standard cell-based and gate-array designs, each output function must be decomposed into a netlist of gates available in the library while preserving speed-independence. This section reviews the complex-gate solution and describes how naive gate-level implementations of output functions can introduce hazards into the circuit, to present an intuitive overview of our new necessary and sufficient conditions that guarantee hazard-freedom.

3.1 Complex-gate implementations

The synthesis problem using complex gates amounts to deriving the next-state equations for each output signal from the SG, finding the optimal logic function satisfying the next-state equation, and implementing the optimal logic function with a single complex gate. As described in [12] and [37], the next-state equation of an output signal u is an incompletely specified function derived from the state graph. For each state $s \in \Phi$ in which u is enabled to rise or has already risen and is not enabled to fall, its next state is assigned 1. For all other states in $s \in \Phi$, its next state is assigned 0. The remaining unreachable states in the Boolean space ($\mathcal{B}^{A_{Spec}} - \Phi$) are don't cares. Standard Boolean techniques can be used to optimize the next-state equation [8]. The resulting equation is then assumed to be implemented using a single complex gate. For the output signal c in Figure 1, the next-state function and complex-gate implementation is illustrated in Figures 2(a) and 2(b).

A complex-gate correctly implements a next-state function if the gate evaluates to the next-state function in each state. When every complex gate correctly implements the corresponding next-state function, the circuit is *complex-*

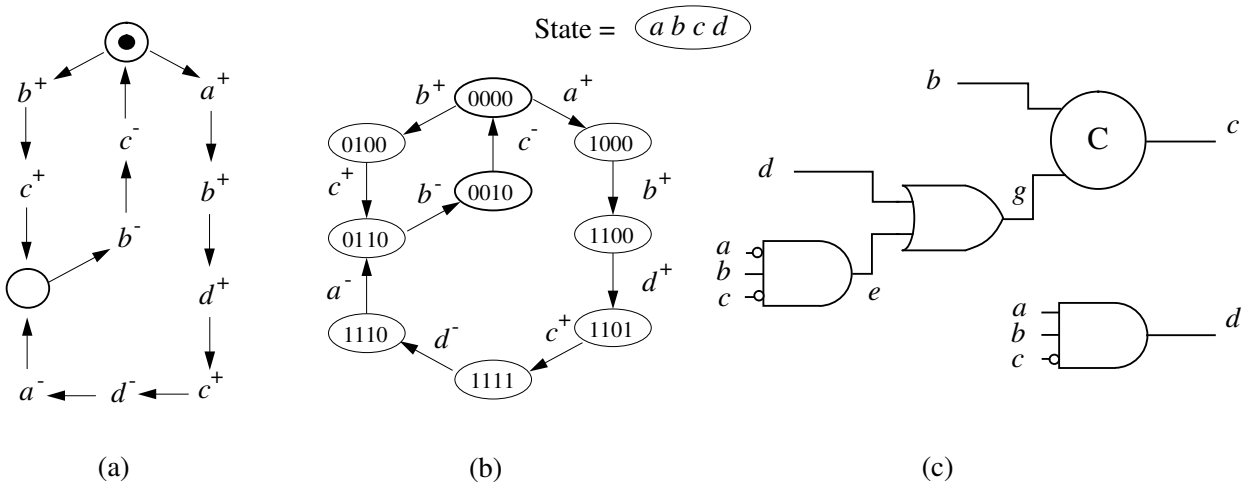


Figure 1: (a) A STG with input choice (motivated by the example STG in Figure 3(d) of [13]), (b) the corresponding determinate SG and, (c) a corresponding hazard-free gate-level circuit.

enabled. Complete state coding is a necessary property for a state graph to be implementable as a speed-independent circuit [12]. It has been reported that adding state variables can transform an arbitrary SG into one that satisfies complete state coding [13, 28, 51]. We therefore assume that one of these algorithms is used before the SG is fed into our synthesis tool.

A SG with complete state coding can be transformed into a SG with unique state coding by folding states with the same label into a single state. More precisely, to fold states, take every pair of states s and s' with the same label, remove s' from the set of states Φ, and replace s' with s in every place s' appears in Γ. Also, if $s_0 = s'$, reassign s_0 to s. Note that folding such states has no effect on the semantics of the specified output signal behavior and guarantees that states can be unambiguously referred to by their label, thereby simplifying algorithm presentation. Therefore, without loss of generality, our algorithms assume that the state graph has unique state coding.

A signal u is disabled by a signal v, $v \neq u$, in a state s if u is enabled in state s and not enabled in s' where $s \xrightarrow{v} s'$. A SG is *determinate speed-independent* if in every state transition in the SG, output signals may not disable any signals (no output choice) and input signals may not disable output signals but may disable other input signals (input choice).

The SG in Figure 1(b) is determinate speed-independent. Formally, this SG is modeled by $\langle I, O, \Phi, \Gamma, s_0, \lambda \rangle$ where $I = \{a, b\}$ and $O = \{c, d\}$. There are 9 states in the set of states Φ including states [0000] and [0100]. There are 9 transitions in the set of transitions Γ including [0000] $\xrightarrow{a}$ [1000] and [0000] $\xrightarrow{b}$ [0100]. These two state transitions illustrate input choice between a and b since when a fires, b is disabled and when b fires, a is disabled.

2.2 Circuit implementation

A circuit implementation is modeled by a graph $\langle I, O, N, E, F \rangle$, where I is the set of input signals, O is the set of output signals, N is the set of internal signals, E is a set of connections between signals, and F is a set of local gate functions. The set of *circuit signals* is the union of inputs, outputs, and internal signals (denoted A_{Impl}). Each edge $e \in E$ represents a connection between circuit signals. An edge e is directed, connecting a source signal to a sink signal. The set of fanins of u, denoted $FI(u)$, is all sources of edges that have u as its sink. Similarly, the set of fanouts of u, denoted $FO(u)$, is all sinks of edges that have u as its source. This paper considers only *externally-cut* circuits in which outputs of all memory elements are primary outputs and external signals cut the circuit.

The status of an implementation at some point in time is modeled using an *implementation state* that is either a bitvector over A_{Impl}, i.e., $q \in 2^{A_{Impl}}$, or the special value, q_{fail}, that models the failure state of an implementation entered after the occurrence of a *hazard* (defined formally in Appendix E). Each gate output signal u has an associated Boolean function $f \in F$ of arity $|FI(u)| + 1$. We refer to $f_u(q(u), q(v_1), \ldots, q(v_r))$, where $FI(u) = \{v_1, \ldots, v_r\}$, as the *internal evaluation* of u in q since it depends on the values of circuit signals in state q and we abbreviate it as $f_u(q)$.

very concurrent specifications sometimes require using high-fanin and/or complex gates. Therefore, the second step of the procedure heuristically decomposes all high-fanin and/or complex gates into limited-fanin basic gates. This decomposition often requires the addition of connections between gates to remove any hazards caused by the added delay elements associated with the introduction of new internal signals. Third, the procedure applies a variety of logic optimizations that improve circuit area and delay.

Both the decomposition and logic optimization procedures explore logic transformations that are not guaranteed to preserve hazard-freedom. Hence, before accepting any logic transformation, the procedure verifies that the transformed circuit is hazard-free. Circuit verification has traditionally been computationally expensive, and our synthesis procedure may require hundreds of verification subtasks. Therefore, to reduce the complexity of the verification subtasks the procedure uses a logic transformation framework that utilizes *observability don't cares* (ODCs) and *incremental verification*. This framework dramatically reduces the amount of computation required for verifying a sequence of logic transformations, thereby making our synthesis procedure computationally feasible.

We present the area and performance results of the synthesized circuits for all benchmark specifications given in [27]. Our tool successfully synthesized all circuits with limited-fanin (4-input or less) basic gates. We compare our synthesized circuits with their non-speed-independent counterparts synthesized by the asynchronous circuit synthesis package provided in Berkeley's SIS, in which bounded delay lines are used to remove circuit hazards. The comparison shows that our circuits are on average 14% larger but 13% faster. These results suggest that while our constrained logic synthesis strategy to achieve speed-independence may cost marginally in area, the lack of bounded delay lines often yields faster, more robust circuits than can otherwise be achieved.

2 Specification and implementation

Automated circuit synthesis begins with a high-level specification that describes the desired input/output behavior of the circuit. For asynchronous circuits, many styles of high-level specifications have been proposed including languages [32, 50], graphs [12, 14], and state machines [43]. Although there are many differences between the various specification styles, they all model (at least implicitly) both the desired circuit behavior and the circuit's environment. Thus, unlike many synchronous specification styles, these specifications dictate the input behavior the circuit can expect, without which it would be quite difficult, if not impossible, to synthesize a hazard-free implementation. An example of a simple specification using a *signal transition graph* STG with choice [12] is given in Figure 1(a).

The first step in our synthesis strategy is to translate the high-level specification into an intermediate format called a state graph (SG). A SG is general enough to capture all external behavior of a system and can be easily obtained from most high-level languages. The SG obtained from the STG depicted in Figure 1(a) is depicted in Figure 1(b). From the SG, our synthesis procedure synthesized a circuit that implements all output signals with a netlist of basic gates such as AND gates, OR gates, and C-elements with arbitrary inverted inputs and number of fanins limited by a given maximum. A hazard-free gate-level speed-independent circuit for the SG depicted in Figure 1(b) is depicted in Figure 1(c). This section formalizes both our state graph and circuit implementation models.

2.1 State graph

A state graph (SG) is modeled by the tuple $\langle I, O, \Phi, \Gamma, s_0, \lambda \rangle$, where I is a set of input signals, O is a set of output signals, Φ is a set of states, $\Gamma \subseteq \Phi \times \Phi$ is a set of state transitions, s_0 is the initial, or *reset*, state, and λ is a labeling function for states. The set of *external signals* is the union of the input and output signals (denoted A_{Spec}).

The labeling function λ labels each state $s \in \Phi$ with a bitvector over the external signals, i.e., $\lambda(s) \in \mathcal{B}^{A_{Spec}}$. The value of a signal $u \in A_{Spec}$ in a state s is the value of u in the label of s, i.e., $s(u) \equiv \lambda(s)(u)$. The function *bitcomp*$(s, u)$ returns the label formed from s by complementing the bit corresponding to u. For example, for the state s labeled [0000] in the SG in Figure 1(b), *bitcomp*$(s, a) = [1000]$.

A state graph must be strongly connected and each ordered pair (s, s') must differ in exactly one signal, i.e., $\lambda(s') =$ *bitcomp*(s, u) for some $u \in A_{Spec}$. For a state transition $(s, s') \in \Gamma$ in which $\lambda(s') =$ *bitcomp*(s, u), the notation $s \xrightarrow{u} s'$ may be used. A signal $u \in A_{Spec}$ is enabled in state s (denoted *enabled*(u, s)) if u can change in state s, that is, if there exists an $s' \in \Phi$ such that $s \xrightarrow{u} s'$ holds.

A SG has *unique state coding* [12] if every state in the state graph has a unique label. A SG has *complete state coding* [12] if for every pair of states s and s' that have the same label, s and s' have the same output signals

1 Introduction

As competitive asynchronous chips gain attention [10, 15, 20, 31, 42, 47, 49], asynchronous design is increasingly being considered as a practical and efficient design alternative. Asynchronous designs do not require a global clock for synchronization. Instead, synchronization is event-driven that is transitions on wires act to request the start of a computation and acknowledge its completion. By removing the global clock, this design style has several advantages including absence of problems related to clock-skew, freedom from designing for worst-case delay, and automatic power-down of unused circuitry.

Speed-independent circuits are an attractive subclass of asynchronous circuits because they can tolerate delay variations resulting from variations in IC processing, temperature, and voltage. More precisely, these circuits work correctly regardless of the delays of individual gates, while assuming constant wire delays for multiple fanouts [39]. As a result, achieving speed-independence avoids the need for timing assumptions and delay lines that tend to increase area and delay while reducing the reliability of the circuit. This insensitivity to variation in gate delays also means that components within a speed-independent system can be replaced by faster components without needing to redesign any other part of the system. Moreover, speed-independent circuits can exhibit more concurrency than fundamental mode circuits [43, 48] which require that inputs change only after the entire circuit is guaranteed to be stable. Speed-independent circuits can also be easily verified [3, 18] and have a testability advantage—they are self-checking with respect to a broad class of multiple output stuck-at-faults [2].

Previously, automated synthesis of speed-independent circuits required complex-gates—every specified output signal was implemented with a single, possibly very complicated, atomic gate [12, 37]. Unfortunately, such circuits can be unreliable, since unmodeled glitches within the complex-gates may cause circuit malfunction. These unmodeled glitches are especially problematic in standard-cell and programmable gate-array implementations in which complex-gates are usually implemented with a collection of standard logic cells. A more reliable approach is to synthesize gate-level implementations comprised of only basic gates that can be easily incorporated into standard-cell and gate-array libraries.

To build gate-level speed-independent circuits, Martin and Burns add state variables to the specification to simplify all complex-gates to gates that exist in the gate library or can be reliably generated using a module generator [33]. Unfortunately, the addition of state variables often requires user intervention and some circuits require specialized gates which may not be suited for gate-array and standard-cell implementations. Nevertheless, this semi-automated method has been used to build many large custom designs including a sixteen-bit asynchronous microprocessor [34, 47].

Kishinevsky and Varshavsky proved that a gate-level speed-independent implementation can always be found for a limited class of specifications (Chapter 5 of [52]). They considered *distributive* specifications of autonomous circuits [38], which have no inputs. They proved that all such specifications can be implemented speed-independently using 2-input NAND gates. Their goal, however, was theoretical in nature and did not include practical considerations such as speed and area. As a result, their algorithm produces large, complex circuits because it unnecessarily adds many state variables to avoid hazards. In addition, since their circuits do not model inputs, their algorithm is restricted to circuits that do not exhibit conditional behavior modeled by input (environmental) choice.

Since then, there have been several works on synthesizing speed-independent circuits from specifications with choice to basic gates with unlimited fanin. First, in a preliminary version of this work [5], we developed an algorithm to generate unlimited-fanin gate-level speed-independent circuits from state graph (SG) specifications that can model input choice. Subsequently, K. Lin and C. Lin developed an algorithm transforming a free-choice signal transition graph (STG) into an unlimited-fanin gate-level speed-independent circuit [30]. In addition, Kondratyev et al. proposed a similar synthesis algorithm to our previous work [25]. Both the works of [30] and [25] do not address the problem of synthesis using limited-fanin gates and restrict the circuit's combinational blocks to two-level logic which may not be the optimal implementation. In addition, the theory in [25] is ambiguous, allowing some hazardous circuits to be synthesized. Fortunately, this ambiguity can be resolved with an amendment [24], making their theory essentially equivalent to our results presented in [5].

This paper describes a CAD tool for the synthesis of gate-level speed-independent circuits from a class of SG specifications that include input choice. Our synthesis procedure differs from those described in [30] and [25] in that it addresses synthesis using only limited-fanin gate libraries, and it is capable of using multi-level logic when necessary. Allowing multi-level implementations can increase gate sharing, reducing circuit area, and can shorten the critical path, increasing circuit speed.

The synthesis procedure has three steps. First, the procedure implements the circuit using a *standard C-implementation* logic template. The procedure employs efficient *binate covering algorithms* to find the optimal circuit using this template. While this circuit often can be implemented with limited-fanin basic gates, larger or

C The cube approximation **30**
C.1 Mapping the cube approximation to a set of states 31
C.2 Required fanins and acknowledgement 31

D Sufficient conditions for correctness **32**

E Proof of correctness **34**
E.1 Defining a cube approximation fixpoint 34
E.2 Acknowledgement and stability 35
E.3 Final proof of our synthesis algorithm 35

Contents

1 Introduction **1**

2 Specification and implementation **2**
2.1 State graph . . . 2
2.2 Circuit implementation . . . 3

3 The gate-level synthesis problem **4**
3.1 Complex-gate implementations . . . 4
3.2 Gate-level implementations . . . 5
3.2.1 The request/acknowledge protocol . . . 5
3.2.2 Acknowledgement and monotonicity . . . 5

4 Standard C-implementation **7**
4.1 Theory . . . 7
4.1.1 Excitation and quiescent regions . . . 7
4.1.2 Correct covers . . . 8
4.2 Algorithms . . . 9
4.2.1 General algorithm . . . 9
4.2.2 Single-cube algorithm . . . 11

5 Implementing region functions **13**
5.1 Acknowledgement wire forks . . . 14
5.2 Algorithm for repairing a decomposition . . . 14

6 Logic optimizations **17**
6.1 Removing redundant gate inputs . . . 17
6.2 Sharing and merging gates . . . 17
6.3 Signal network simplification . . . 17
6.4 Inverter bubble shuffling . . . 18

7 Performing logic transformations and optimizations **18**
7.1 Overview . . . 18
7.2 Incrementally verifying complex-gate equivalence with ODCs . . . 18
7.2.1 Representation . . . 19
7.2.2 Sensitized and insensitized signals . . . 19
7.2.3 Finding ODC's . . . 20
7.2.4 Using ODC's to perform logic transformations . . . 20
7.2.5 An example . . . 21
7.3 Incrementally verifying hazard-freedom . . . 21
7.3.1 The cube approximation and sufficient conditions for correctness . . . 21
7.3.2 Localizing the effects of a logic transformation . . . 22
7.4 Accepting logic transformations . . . 22

8 Results and conclusions **22**
8.1 Breaking up complex and high-fanin gates . . . 23
8.2 Comparison with SIS . . . 24
8.3 Summary . . . 24

A Definition of Correctness **28**
A.1 Complex-gate equivalence . . . 28
A.2 Hazard-freedom . . . 28

B Necessary and sufficient conditions for correctness **29**
B.1 Formalization of monotonicity and acknowledgement . . . 29
B.2 Relationship between monotonicity/acknowledgment and successful implementation states . . . 30

Copyright ©1994

Peter A. Beerel, Chris J. Myers, and Teresa H.-Y. Meng

AUTOMATIC SYNTHESIS OF GATE-LEVEL SPEED-INDEPENDENT CIRCUITS

Peter A. Beerel*, **Chris J. Myers, and Teresa H.-Y. Meng**

Technical Report: CSL-TR-94-648

December, 1994

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305-4055

Abstract

This paper presents a CAD tool for the synthesis of robust asynchronous control circuits using limited-fanin basic gates such as AND gates, OR gates, and C-elements. The synthesized circuits are speed-independent; that is, they work correctly regardless of individual gate delays. Included in our synthesis procedure is an efficient procedure for logic optimizations using *observability don't cares* and *incremental verification*. We apply the procedure to a variety of specifications taken from industry and previously published examples and compare our speed-independent implementations to those generated using a non-speed-independent synthesis procedure included in Berkeley's SIS. Our implementations are not only more robust to delay variations since those produced by SIS rely on bounded delay lines to avoid circuit hazards but also are on average 13% faster with an area penalty of only 14%.

Key Words and Phrases:

Computer-aided design, asynchronous circuit synthesis, gate-level speed-independent circuits, state graph specifications, standard C-implementations, hazard-preserving logic transformations, acknowledgement and monotonicity, binate covering, multi-level logic, conservative incremental verification, and observability don't cares.

*Peter A. Beerel is now an affiliated with the EE-Systems Dept., University of Southern California, Los Angeles, CA 90089-2562

AUTOMATIC SYNTHESIS OF GATE-LEVEL SPEED-INDEPENDENT CIRCUITS

Peter A. Beerel
Chris J. Myers
Teresa H.-Y. Meng

Technical Report: CSL-TR-94-648

December, 1994

This research was supported in part by ARPA, contract DABT63-91-K-0002,
the Center for Integrated Systems, Stanford University, and an NSF Fellowship.