

# Efficient Algorithms for Exact Two-Level Hazard-Free Logic Minimization

Hans M. Jacobson and Chris J. Myers

**Abstract**— This article presents a new approach to two-level hazard-free logic minimization in the context of extended burst-mode finite state machine synthesis. The approach achieves fast single output logic minimization that yields solutions that are exact in number of literals. This article presents algorithms and hazard constraints targeting both generalized C-element and two-level standard gate implementations. The logic minimization approach presented in this article is based on state graph exploration in conjunction with single-cube cover algorithms. The algorithm achieves fast logic minimization by using compacted state graphs, cover tables, and a divide and merge algorithm for efficient single output minimization. The exact two-level hazard-free logic minimizer presented in this article finds minimal number of literal solutions and is several orders of magnitude faster than existing literal exact methods for the largest benchmarks available to date. This includes a benchmark that has never been possible to solve exactly in number of literals before.

**Keywords**— Logic minimization, two-level, gC, hazard-free, extended burst-mode, compacted state graphs, finite state machines, asynchronous.

## I. INTRODUCTION

In recent years, there has been a number of successful and practical asynchronous circuits designed using asynchronous finite state machines in the form of extended burst-mode controllers [4], [19], [24], [28], [32]. During the design process, it is necessary to try different protocols and state assignments to find a good circuit implementation. This often leads to many iterations through the synthesis process. The bottleneck of finite state machine synthesis is typically in the two-level hazard-free logic minimization step. Since existing exact synthesis methods are slow, and may not even complete for large circuits, designers are often forced to use heuristic methods, or partitioning, to interactively explore the design space in a reasonable amount of time often yielding sub-optimal circuits. This article describes a new exact single output two-level hazard-free logic minimization method that utilizes fast algorithms based on state graph exploration to produce solutions for extended burst-mode controllers in a fraction of the time of the state-of-the-art tools. This new method yields literal exact solutions even for a controller that previously could not be solved exactly in number of literals.

In recent designs, such as Yun’s differential equation solver [32] and the burst-mode portions of Intel’s RAP-

PID [28], it has been found that substantial performance improvements can be achieved through the use of generalized C-element (gC) circuit implementations over two-level standard gate implementations. The limitation of targeting gC circuit implementations is that it requires specialized gCs to be in the gate library. Such cells are not typically found in standard-cell or gate-array libraries, and may need to be specially designed. In order to facilitate efficient semi-custom design of integrated circuits, it is necessary to synthesize to standard gates such as ANDs and ORs. The key difficulty in doing so for asynchronous controllers is the need to avoid introducing *hazards*. A hazard is the potential for an unwanted signal transition or glitch. Glitches in asynchronous design typically lead to circuit failure, so hazards must be avoided to guarantee correct circuit operation. In restricting the library to compact, atomic gCs, most hazard concerns are avoided. In particular, the only hazard issue that needs to be addressed is for *dynamic*  $0 \rightarrow 1$  output transitions. However, when targeting a standard gate implementation, additional hazards for *static*  $1 \rightarrow 1$  and *dynamic*  $1 \rightarrow 0$  transitions may also occur. The algorithms presented in this article support both gC and standard gate implementations.

In [33], state minimization and state assignment methods are developed for extended burst-mode controllers. One method is based on excitation region covers allowing for gC circuit implementations while another method is based on ON-set region covers allowing for standard gate implementations. These synthesis methods have been implemented in the 3D tool [31]. 3D leverages the HFMIN tool to perform hazard-free logic minimization and the tool described in [10] to perform technology mapping. The hazard-free logic minimization method presented in this article performs the same function as the HFMIN tool in the 3D synthesis flow.

Rather than leveraging standard approaches to burst-mode logic minimization, the approach taken in this article exploits standard synthesis techniques traditionally used for synthesis of speed-independent circuits, namely state graph exploration and derivation of single-cube covers [1], [2], [14], [21]. In this article these algorithms are applied to the synthesis of *extended burst-mode state machine controllers* targeting *generalized C-element* (gC) as well as *standard gate* implementations. To reach the goal of better support for interactive exploration of design alternatives our approach introduces two new methods to improve runtime of the traditional algorithms. First, state exploration is combated through *compacted state graphs*. This reduces time spent traversing the state graph and also produces smaller cover tables. Second, the problem of finding a minimal cover for an output is solved through di-

This research is supported by a U.of Utah Graduate Research Fellowship, NSF awards MIP-9988329 and MIP-9625014, SRC contract 97-TJ-487, and a grant from Intel Corporation.

H. Jacobson is with the School of Computing, University of Utah, Salt Lake City, UT 84112. E-mail: hans@cs.utah.edu.

C. Myers is with the Dept. of Elec. and Comp. Eng., University of Utah, Salt Lake City, UT 84112. E-mail: myers@ece.utah.edu.

viding the minimization problem into more easily solved sub problems whose results are then merged into a final solution. This is typically faster than solving the whole problem at once when the number of sub problems are limited. The presented work also includes the issues of modeling extended burst-mode hazard constraints and deriving required covers for two-level ON-set implementations in algorithms originally intended for speed-independent excitation region minimization.

Our resulting literal exact minimizer is often several orders of magnitude faster than existing exact minimizers for large benchmarks. It is also competitive in runtime with existing heuristic and cube-exact minimizers while producing covers with up to 25 percent less literals. Finally, our tool achieves sub-second minimization for even the largest gC benchmarks, enabling interactive manipulation of such controllers in real-time.

This article is organized into nine sections. Section II presents previous work in two-level hazard-free logic minimization. Section III gives an overview of terminology used in the logic minimization domain. Section IV presents background in asynchronous logic synthesis. Section V introduces the compacted state graph model which is an efficient means of representing highly concurrent controllers. Section VI describes our new logic minimization algorithm that addresses the hazard issues present in standard gate realizations. Section VII describes the algorithm as it is adapted to address hazards present in generalized-C element realizations. Finally, Section VIII presents our experimental results, and Section IX gives our conclusions.

## II. PREVIOUS WORK

The state-of-the-art in literal exact two-level hazard-free logic minimization is HFMIN [6]. This tool is fast for small to medium sized benchmarks. However, runtime quickly increases with larger controllers. The tool IMPYMIN [29], which is based on implicit algorithms, performs substantially better runtime wise for large benchmarks. IMPYMIN produces cube-exact solutions. The most time efficient tool to date is ESPRESSO-HF [29]. It uses a heuristic algorithm that produces good covers on average. These minimizers are currently used to perform logic minimization in the 3D [31] and MINIMALIST [7] burst-mode synthesis tools.

A two-level hazard-free logic minimization method based on a divide and conquer approach was proposed in [27]. The method combines the search for prime implicants and finding hazard-free implicants into one algorithm. The method reduces the search space in finding hazard-free prime implicants by finding only useful implicants.

Some researchers [5], [30] have proposed to use speed-independent methods for logic minimization of burst-mode gC controllers by translating them into Petri-net specifications. The generated solutions are hazard-free under the constrained gC decomposition rules imposed by the relaxed burst-mode hazard model presented in [33]. However, these methods do not provide support for extended burst-mode controllers or covers for standard gate implementations.

Structural logic minimization methods to combat the

problem of state explosion in state graph traversal have been explored [11], [18], [26]. These methods perform logic minimization while traversing the input Petri-net specification avoiding generation of potentially large state graphs. These methods, however, may experience a substantial increase in Petri-net size for non-monotonic level signals (frequently used in extended burst-mode specifications). Current implementations do not support extended burst-mode controllers and two-level standard gate implementations.

Recent efforts have been made to produce optimal output solutions over all state reductions and encodings [7], thus avoiding manual design exploration of different state assignments. While these methods are intriguing, the sheer algorithm complexity currently limits the size of controllers that can be automatically explored.

## III. LOGIC MINIMIZATION TERMINOLOGY

This section presents the common logic minimization terminology that is used throughout this article. An *incompletely specified Boolean function*  $f$  of  $n$  variables  $x_1, x_2, \dots, x_n$  is a mapping:  $f : \{0, 1\}^n \rightarrow \{0, 1, -\}$ . Each element  $m$  of  $\{0, 1\}^n$  is called a *minterm*. The value of a variable  $x_i$  in a minterm  $m$  is given by  $m(i)$ . The *ON-set* of  $f$  is the set of minterms which return 1. The *OFF-set* of  $f$  is the set of minterms which return 0. The *don't care (DC)-set* of  $f$  is the set of minterms which return  $-$ .

A *literal* is either the variable,  $x_i$ , or its complement,  $\bar{x}_i$ . The literal  $x_i$  evaluates to 1 in the minterm  $m$  when  $m(i) = 1$ . The literal  $\bar{x}_i$  evaluates to 1 when  $m(i) = 0$ . A *product* is a conjunction (AND) of literals. A product evaluates to 1 for a given minterm if each literal evaluates to 1 in the minterm, and the product is said to *contain* the minterm. A set of minterms which can be represented with a product is called a *cube*. A product  $Y$  contains another product  $X$  (i.e.,  $X \subseteq Y$ ) if the minterms contained in  $X$  are a subset of those in  $Y$ . The *intersection* of two products is the set of minterms contained in both products. A *sum-of-products* (SOP) is a set of products that are disjunctively combined. In other words, a SOP contains a minterm when one of the products in the SOP contains the minterm.

An *implicant* of a function is a product that contains no minterms in the OFF-set of the function. A *prime implicant* is an implicant which is contained by no other implicant. A *cover* of a function is a SOP which contains the entire ON-set and none of the OFF-set. A cover may optionally include part of the DC-set. The two-level logic minimization problem is to find a minimum-cost cover of the function. Ignoring hazards, a minimal cover is always composed only of prime implicants.

## IV. ASYNCHRONOUS LOGIC SYNTHESIS BACKGROUND

For asynchronous design, the two-level logic minimization problem is complicated by the fact that there must be no hazards in the circuit implementation. As mentioned previously, the existence of a hazard indicates that under some assignment of delays to the gates and wires, an unwanted signal transition may occur. This can cause catastrophic consequences for an asynchronous design. Consider

such a glitch occurring on a wire that serves as a request to a function unit to perform an operation. This glitch may be interpreted as a request when none was intended putting the system in an unacceptable state. This section describes our assumptions and hazard models for gC and standard gate implementations, and a discussion of the requirements necessary to eliminate all hazards.

*A. Extended burst-mode machines*

The synthesis method described in this article produces hazard-free implementations for controllers specified as *extended burst-mode* (XBM) machines, a class of multiple input change (MIC) asynchronous finite state machines. Inputs are allowed to change concurrently in a form of MIC called *bursts*. The signals within a burst may arrive in arbitrary order. Every *input burst* is followed by a (possibly empty) concurrent burst of output and state signal changes. After the output and state burst, the internal nodes of the circuit are allowed to stabilize before the fed back state signals are allowed to reach the inputs of the circuit. The circuit operates under *fundamental-mode* which means that the circuit must be allowed to stabilize in response to the fed back state signals before the next input burst can arrive. The extended burst-mode specification shown in Figure 1 is used as an example in later sections. A transition between states can be performed in three ways. A normal transition contains an input burst followed by a possibly empty concurrent state and output burst, also called Type I machine behavior [33]. Alternatively, an input burst can be followed by a sequential output and state burst, a Type II machine, or by a sequential state and output burst, a Type III machine [33]. An XBM machine typically make use of Type I transitions. However, Type III behavior is necessary to avoid dynamic hazards in two-level standard gate implementations due to non-monotonic level signals (e.g. see transition between states 5 and 6 in Figure 1). If there is more than one outgoing transition from a state, a deterministic choice is implied. Signals annotated with + and - signs and not enclosed in square brackets imply a rising and falling transition, also known as a *terminating edge*. Input signals annotated with a \* are called *directed don't cares* and are free to change monotonically at any time during a sequence of specified directed don't care states but must change by the time the signal is next specified as a terminating edge. A terminating edge not preceded by a directed don't care is called a *compulsory edge*. Signals enclosed within square brackets are *conditionals* (level signals) and are free to change non-monotonically whenever not specified. When specified, such a conditional is assumed to reach a stable value in time to be sampled correctly by the arriving compulsory edges of the burst. State transitions occur only when all conditionals are met and all terminating edges pertaining to a burst have appeared.

The following three timing requirements are assumed in the extended burst-mode synthesis method.

- *Fundamental-mode environmental constraint*: no compulsory edge of the next input burst may appear until the circuit has attained quiescence.

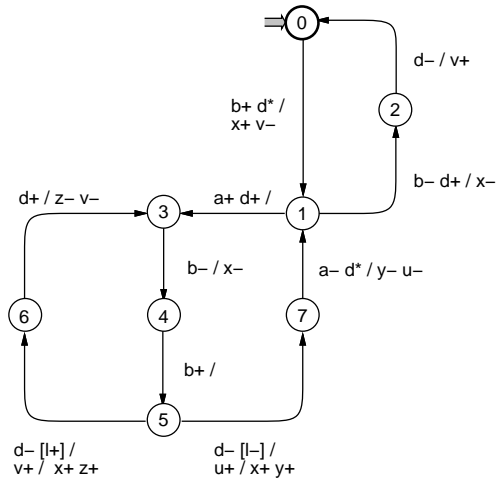


Fig. 1. Extended burst-mode controller *ack-xbm-si*.

- *Feedback delay requirement*: fed back variable changes must not reach the inputs until all enabled outputs and state variables have completed.
- *Setup and hold time requirements*: all conditionals (level signals) specified to be stable must stabilize before any compulsory edge appears and must remain stable until the output and state burst has been completed.

A transition in an XBM machine begins in one cube  $c_1$  and ends in another cube  $c_2$  where the values of multiple variables may change during the transition. The cube  $c_1$  is called the *start cube* and  $c_2$  is called the *end cube* of the transition. The smallest cube that contains both  $c_1$  and  $c_2$  is called the *generalized transition cube* and is denoted  $[c_1, c_2]$  [33]. This cube includes all possible minterms that a machine may pass through starting in  $c_1$  and ending in  $c_2$ . A generalized transition cube can also be represented with a product which contains a literal for each variable  $x_i$  in which  $c_1(i) = c_2(i) \neq -$ . An *open generalized transition cube*  $[c_1, c_2]$  includes all minterms in  $[c_1, c_2]$  except those in  $c_2$ . An open transition cube usually must be represented using a set of products.

In a generalized transition cube, signals may be of type rising, falling, or level. Rising and falling signals change monotonically (i.e., at most once in a legal generalized transition cube). Level signals must hold the same value in  $c_1$  and  $c_2$ , where the value is either a constant (0 or 1) or a don't care (-). Level signals, if they are don't care, may change non-monotonically.

*B. Implementation structures*

*Standard gate implementation structure*. In an extended burst-mode standard gate implementation of the output function  $f$ , the ON-set of  $f$  is implemented as a two-level SOP circuit. The product terms are implemented with AND gates, and the sum is implemented as an OR gate. Figure 2(a) illustrates the structure of an extended burst-mode standard gate circuit.

*gC implementation structure*. In an extended burst-mode gC implementation of the output function  $f$ , the set logic ( $f_{set}$ ) and reset logic ( $f_{reset}$ ) are both implemented as

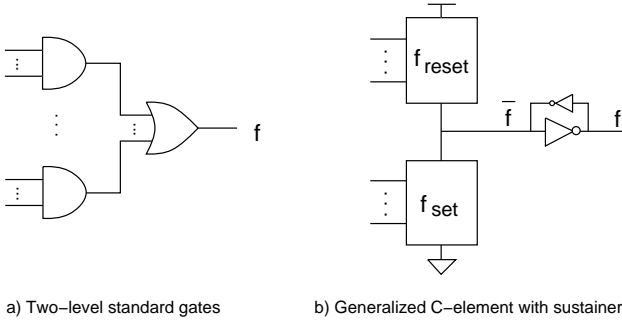


Fig. 2. Two-level standard gate and gC implementations.

two-level SOP circuits. The set logic  $f_{set}$  is implemented by the N-mos transistor network and the reset logic  $f_{reset}$  is implemented by the P-mos transistor network. The arguments for hazard-free covers are analogous for  $f_{set}$  and  $f_{reset}$ . Figure 2(b) illustrates the structure of an extended burst-mode gC circuit.

### C. Hazard models

If a function  $f$  does not change monotonically during a MIC, then  $f$  has a *function hazard* for that transition. If a transition has a function hazard, there exists no implementation of the function which avoids the hazard during the transition. In an XBM machine, the types of transitions are restricted such that each function may only change value after the completion of an input burst. A generalized transition  $[c_1, c_2]$  for a function  $f$  is an *extended burst-mode transition* if for every minterm  $m_i \in [c_1, c_2]$ ,  $f(m_i) = f(c_1)$  and for every minterm  $m_i \in c_2$ ,  $f(m_i) = f(c_2)$ . Therefore, if a function only has extended burst-mode transitions, then it is function hazard-free [22], [33].

Although there are no function hazards, there may be *logic hazards*. In order to design a hazard-free SOP cover, we must consider each possible type of transition in turn.

First, if during a static  $1 \rightarrow 1$  transition  $[c_1, c_2]$ , the cover of  $f$  can momentarily evaluate to 0, then there exists a static 1-hazard. In a standard gate SOP cover, consider the case where there is one product  $p_1$  which contains  $c_1$  but not  $c_2$  and another product  $p_2$  which contains  $c_2$  but not  $c_1$ . The cover includes both  $c_1$  and  $c_2$ , but there can be a static 1-hazard. If  $p_1$  is implemented with a faster gate than  $p_2$ , then the gate for  $p_1$  may turn off faster than the gate for  $p_2$  turns on which can lead to the cover momentarily evaluating to a 0, thus exhibiting a momentary  $1 \rightarrow 0 \rightarrow 1$  glitch on the output. When multiple variables are changing concurrently, the cover may pass through other minterms along the way between  $c_1$  and  $c_2$ . To be free of static 1-hazards, it is necessary that a single product in the cover include all these minterms. In other words, each generalized transition cube,  $[c_1, c_2]$ , where  $f(c_1) = f(c_2) = 1$ , must be contained in some product in the cover to eliminate static 1-hazards.

In contrast, a static 1-hazard cannot occur in a gC SOP cover. The products  $p_1$  and  $p_2$  are both implemented as transistor stacks in the gC gate. The output of the gC gate is held stable by a weak feedback staticizer whenever no stack in the gC gate is conducting. Subsequently no

hazard will occur even if the stack for  $p_1$  turns off before the stack for  $p_2$  turns on.

In a SOP cover of a function, the only way there can be a static 0-hazard is if some product includes both signal,  $x_i$ , and its complement  $\bar{x}_i$ . Clearly, such a product would not be included in a minimal SOP cover. Subsequently SOP covers for both two-level and gC implementations never produce a static 0-hazard.

Dynamic hazards are a little more complicated, and additional definitions are needed. The *start subcube*,  $c'_1$ , is a maximal subcube of  $c_1$  such that each signal undergoing a directed don't care transition is set to its initial value (i.e., 0 for a rising transition and 1 for a falling transition). The *end subcube*,  $c'_2$ , is a maximal subcube of  $c_2$  such that each signal undergoing a directed don't care transition is set to its final value (i.e., 1 for a rising transition and 0 for a falling transition).

For each dynamic  $1 \rightarrow 0$  transition,  $[c_1, c_2]$ , if a product in a standard gate SOP cover intersects  $[c_1, c_2]$  (i.e., it includes a minterm from the transition), then it must also include the start subcube,  $c'_1$ . For each dynamic  $0 \rightarrow 1$  transition,  $[c_1, c_2]$ , if a product in a standard gate or gC SOP cover intersects  $[c_1, c_2]$ , then it must also include the end subcube,  $c'_2$ . If a product term contains  $c'_1$  or  $c'_2$ , then it changes value monotonically during the transition. If a product term does not contain  $c'_1$  or  $c'_2$  but still intersects the transition cube  $[c_1, c_2]$ , then it may change value non-monotonically during the transition. During the transition from  $c_1$  to  $c_2$ , the product term starts out as 0, then evaluates to 1 when the transition crosses the intersection point, and then goes back to 0 once the transition has passed the intersection point. For a dynamic  $1 \rightarrow 0$  transition this could result in a  $1 \rightarrow 0 \rightarrow 1 \rightarrow 0$  glitch if the intersecting product term is slow to turn on in a standard gate cover. Similarly, a dynamic  $0 \rightarrow 1$  transition could result in a  $0 \rightarrow 1 \rightarrow 0 \rightarrow 1$  glitch if the intersecting product term turns on and off quickly [33]. Hazards for dynamic  $0 \rightarrow 1$  transitions are a problem for both standard gate and gC implementations, while hazards for dynamic  $1 \rightarrow 0$  transitions are a problem only in standard gate implementations. The difference in dynamic hazard characteristics between standard gate and gC implementations is due to decomposition restrictions in the gC case [10] that make sure  $1 \rightarrow 0$  hazards are not introduced.

### D. Summary of hazard requirements

The hazard-free cover requirements for the ON-set of a two-level standard gate SOP implementation of output function  $f$  are:

1. Each ON-set cover cube of  $f$  must not intersect the OFF-set.
2. Every static  $1 \rightarrow 1$  transition  $[c_1, c_2]$  in  $f$  must be completely covered by some product.
3. For every dynamic  $0 \rightarrow 1$  transition  $[c_1, c_2]$  in  $f$ ,  $c_2$  must be completely covered by some product.
4. For every dynamic  $1 \rightarrow 0$  transition  $[c_1, c_2]$  in  $f$ , each maximal subcube of  $[c_1, c_2]$  must be completely covered by some product.

5. Any product term of  $f$  intersecting a dynamic  $1 \rightarrow 0$  transition  $[c_1, c_2]$  must also contain the start subcube  $c'_1$ .  
 6. Any product term of  $f$  intersecting a dynamic  $0 \rightarrow 1$  transition  $[c_1, c_2]$  must also contain the end subcube  $c'_2$ .  
 Requirements 2, 3, and 4 describe the product terms, also called *required cubes*, that are required for the cover to turn on when it is supposed to. The entire required cube must be covered by some product in the cover to eliminate static hazards. The transition cubes described in requirements 5 and 6 are also called *privileged cubes*. Intersections of these cubes that do not contain the appropriate subcube result in dynamic hazards. Therefore, a hazard-free cover must guarantee there are no such illegal intersections. As shown in Section VI, all of these requirements are satisfied by the logic minimization algorithm presented in this article.

Similarly, the hazard-free cover requirements for the set function,  $f_{set}$ , of a two-level gC SOP implementation of output function  $f$  are:

1. Each set cover cube of  $f_{set}$  must not intersect the OFF-set.
2. For every dynamic  $0 \rightarrow 1$  transition  $[c_1, c_2]$  in  $f$ ,  $c_2$  must be completely covered by some product.
3. Any product term of  $f$  intersecting a dynamic  $0 \rightarrow 1$  transition  $[c_1, c_2]$  must also contain the end subcube  $c'_2$ .

Similarly, the hazard-freedom requirements for  $f_{reset}$  are:

1. Each reset cover cube of  $f_{reset}$  must not intersect the ON-set.
2. For every dynamic  $1 \rightarrow 0$  transition  $[c_1, c_2]$  in  $f$ ,  $c_2$  must be completely covered by some product.
3. Any product term of  $f$  intersecting a dynamic  $1 \rightarrow 0$  transition  $[c_1, c_2]$  must also contain the end subcube  $c'_2$ .

Due to the gC circuit structure and output staticizer there are no static hazard restrictions for gC gates. In addition, there are no restrictions for intersection of dynamic  $1 \rightarrow 0$  transitions for  $f_{set}$  ( $0 \rightarrow 1$  for  $f_{reset}$ ). Hazard-free constraints during decomposition of gC gates [10] makes sure such hazards cannot manifest.

## V. COMPACTED STATE GRAPHS

Our algorithm begins with a *compacted state graph* (CSG). CSGs are an efficient way to represent states of highly concurrent controllers without introducing state explosion. Similar methods to avoid state explosion have been explored for speed-independent property analysis of gate networks [13].

State graphs [3], [20], rather than compacted state graphs, are typically used to represent the reachable state space in logic minimization algorithms based on graph traversal. However, when specifications that have a high degree of signal concurrency and make use of non-monotonic level signals are translated into state graphs, state explosion often occurs. The reason for this state explosion is that each state in a state graph only represents the change of one signal. When multiple signals change concurrently,  $2^n - 1$  states, where  $n$  is the number of signals changing, are required to represent all states that are reachable during the signal changes. Similarly, level signals add another  $2^n$  multiplicative factor to the state space

if such signals are allowed to change non-monotonically. State graph exploration time therefore grows exponentially with the number of level signals and the degree of signal concurrency in the specification. The way states are represented in state graphs typically also carry over to the cover problems in the logic minimization steps, increasing the size of cover tables exponentially.

XBM specifications typically have high signal concurrency and may also include several level signals. Due to state explosion, translating XBM specifications into state graphs may therefore result in very long runtimes for large controllers. By introducing the concept of compacted state graphs in our algorithms, such state explosion is avoided enabling even very large controllers to be traversed and minimized quickly.

A CSG is modeled with a 4-tuple  $\langle I, O, \Phi, \Gamma \rangle$  where  $I$  is a set of input signals,  $O$  is a set of output signals,  $\Phi$  is a set of *compacted states*, and  $\Gamma$  is a set of state transitions. Each compacted state is labeled with a cube consisting of all states reachable by all possible interleavings of the set of currently enabled monotonic (edge) and non-monotonic (level) input and output signals.

For a given compacted state  $s$  and signal  $u$ , we define the function  $s(u)$  as follows:

$$s(u) = \begin{cases} 0 & \text{stable low} \\ R & \text{changing monotonically from 0 to 1} \\ 1 & \text{stable high} \\ F & \text{changing monotonically from 1 to 0} \\ - & \text{changing non-monotonically} \end{cases}$$

CSGs can be easily derived from both BM and XBM state machines. CSGs derived from XBM specifications are composed of compacted states which are one of two different types. A compacted state is said to represent an *input burst* if some inputs are changing while all outputs are stable. A compacted state is said to represent an *output burst* if some outputs are unstable. Due to directed don't cares and level signals, some output bursts may contain unstable inputs. In a CSG derived from an XBM machine, the successor state of an input burst must be either a unique input or output burst state or possibly a set of input burst states (in the case of a choice).

Figure 3 illustrates the translation of a single XBM transition  $a + b + d * / x +$  to its representation as a normal state graph and a compacted state graph. In a compacted state graph it is implicitly assumed that concurrently enabled signals can arrive in any order. All possible interleavings of a set of concurrently enabled signals can therefore be represented as a single compacted state/transition pair. A new compacted state/transition pair is only created when a new signal becomes enabled. In Figure 3 the concurrently enabled signals  $a$ ,  $b$ , and  $d$  represent the input burst of the XBM transition. These enabled signals form the CSG state  $RRR0$ . In the output burst, the directed don't care signal  $d$  remains enabled and output  $x$  becomes enabled. These enabled signals form the CSG state  $11RR$ . In contrast, in a normal state graph, the signal arrival order has to be specified explicitly by modeling all possible interleavings of enabled signals through separate state/transition

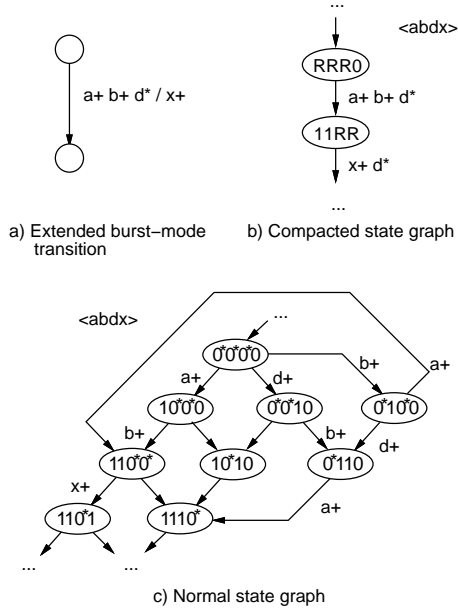


Fig. 3. XBM transition in a compacted and normal state graph.

pairs. As illustrated in Figure 3 the enabled signals  $a$ ,  $b$ , and  $d$  representing the XBM input burst must be modeled as 11 separate state/transition pairs. Signals  $d$  and  $x$  enabled in the output burst must in turn be modeled by 4 state/transition pairs. A significant reduction in state space can be achieved through the use of compacted state graphs. For example, the CSG in Figure 4 has 20 compacted states while a normal state graph would have 88 states. The benefit of CSGs is clear even for an example of this small size. For large XBM controllers the difference in number of states can be several orders of magnitude.

Figure 4 illustrates the CSG derived from the *xbm-si* XBM specification in Figure 1. Consider the translation of the choice modeled in state 1 of the XBM specification. In the CSG the input bursts of the two possible branches are modeled as two compacted states. State *abdl.xyzuv:R1R-.10000* represents the input burst of the transition from XBM state 1 to 3, and *0FR-.10000* represents the input burst of the transition from XBM state 1 to 2. The output burst of the transition from XBM state 1 to 2 is represented by the compacted state *001-.F0000*.

CSGs can be used to represent the set of generalized transitions implementing an XBM function in a compact fashion. In a CSG each compacted state represents a generalized transition. The transition cube for a given compacted state  $s$  can be expressed as  $[initial(s), final(s)]$ , or alternatively as  $cube(s)$ , where  $initial$ ,  $final$ , and  $cube$  can be determined using the following three functions:

$$initial(s)(u) = \begin{cases} 0 & \text{if } s(u) = 0 \text{ or } s(u) = R \\ 1 & \text{if } s(u) = 1 \text{ or } s(u) = F \\ - & \text{if } s(u) = - \end{cases}$$

$$final(s)(u) = \begin{cases} 0 & \text{if } s(u) = 0 \text{ or } s(u) = F \\ 1 & \text{if } s(u) = 1 \text{ or } s(u) = R \\ - & \text{if } s(u) = - \end{cases}$$

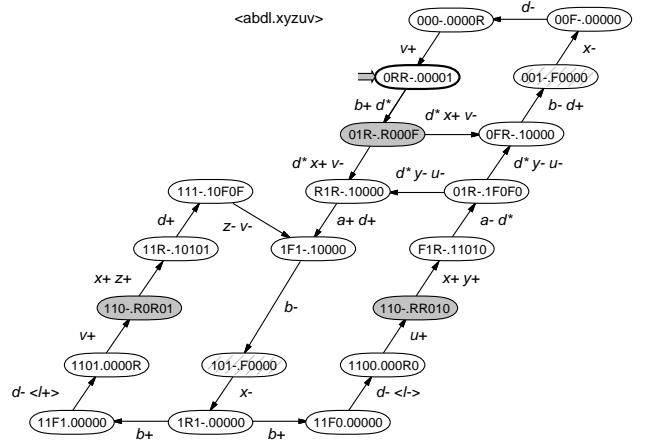


Fig. 4. CSG for controller *ack-xbm-si*.

$$cube(s)(u) = \begin{cases} 0 & \text{if } s(u) = 0 \\ 1 & \text{if } s(u) = 1 \\ - & \text{if } s(u) = - \text{ or } s(u) = R \text{ or } s(u) = F \end{cases}$$

For each output signal  $u$  and each input burst  $s$ , we can determine what type of transition is represented by the transition cube for the state. Assume that  $s(u) = 0$ . First, we must check  $s'$  where  $(s, s') \in \Gamma$ . If  $s'(u) = 0$ , then the transition cube,  $cube(s)$ , is a  $0 \rightarrow 0$  transition. If  $s'(u) = R$ , then  $cube(s)$  is a  $0 \rightarrow 1$  transition. Now assume  $s(u) = 1$ . If  $s'(u) = 1$  then  $cube(s)$  represents a  $1 \rightarrow 1$  transition. Finally, if  $s'(u) = F$ , then  $cube(s)$  is a  $1 \rightarrow 0$  transition. For each output burst  $s$ , the transition cube,  $cube(s)$ , represents the state of signals during the feedback of the output variables back to the inputs. Feedback is always a static transition for an output. Thus if  $final(s)(u) = 1$ , it is a static  $1 \rightarrow 1$  transition, and if  $final(s)(u) = 0$ , it represents a static  $0 \rightarrow 0$  transition.

The *maximal subcubes* for a compacted state are the cubes formed by, for each cube, setting one don't care corresponding to a terminating input edge to its initial value. In order to find the maximal subcubes, we split the state for each such don't care signal using the following function:

$$split(s, v)(u) = \begin{cases} 0 & \text{if } s(u) = 0 \text{ or } (u = v \text{ and } s(u) = R) \\ 1 & \text{if } s(u) = 1 \text{ or } (u = v \text{ and } s(u) = F) \\ - & \text{if } u \neq v \text{ and } (s(u) = - \text{ or } s(u) = R \\ & \text{or } s(u) = F) \end{cases}$$

Table I illustrates the results of the respective functions when applied to state  $s = abdl.xyzuv:0FR-.10000$  of the CSG in Figure 4. In the table,  $v = \{a, b, d, l, x, y, z, u, v\}$ .

TABLE I  
STATE  $s = 0FR-.10000$  APPLIED TO DEFINED FUNCTIONS.

| initial(s) | final(s)   | cube(s)    | split(s,v) |
|------------|------------|------------|------------|
| 010-.10000 | 001-.10000 | 0---.10000 | 01--.10000 |
|            |            |            | 0-0-.10000 |

*Algorithm VI.1:*

```

minimize( $\langle I, O, \Phi, \Gamma \rangle$ )
  foreach  $u \in O$ 
    LocalSol =  $\emptyset$ ;
    RC = required_cubes( $\langle I, O, \Phi, \Gamma \rangle, u$ );
    foreach  $rc \in RC$ 
      tc = trigger_cube( $\langle I, O, \Phi, \Gamma \rangle, u$ );
      CS = context_signals( $I, O, rc, tc$ );
      CV = cover_violations( $\langle I, O, \Phi, \Gamma \rangle, u, tc$ );
      IV = intersection_violations( $\langle I, O, \Phi, \Gamma \rangle, CS, u, rc, tc$ );
      table = build_binate_table(tc, CS, CV, IV);
      LocalSol = LocalSol  $\cup$  solve_binate_table(table, RC);
    table = build_unate_table(RC, LocalSol);
    Sol(u) = solve_unate_table(table);
  return(Sol);

```

Fig. 5. Minimization algorithm for standard gates.

## VI. STANDARD GATE MINIMIZATION ALGORITHM

Our logic minimization algorithm uses state graph traversal and ideas similar to those in the fast single-cube algorithm [21] designed for speed-independent and timed circuits. An important property of generalized transitions is that a required ON-set cube can always be represented by a single cover cube. Therefore, taking a single-cube approach is guaranteed to always find an optimal solution for two-level implementations. Figure 5 presents our logic minimization algorithm for standard gate implementations (our algorithm for gC implementations is presented in the next section). The input to our logic minimization algorithm is a CSG derived from a state assigned XBM machine, and the output is a minimum literal logic equation for each output signal implementing the ON-set regions in the CSG. The logic minimization algorithm begins by traversing the CSG to derive the set of required cubes necessary to cover the ON-set. The algorithm then divides up the minimization problem into several subproblems, one for each required cube, that are solved separately. For each subproblem, the algorithm then proceeds to find the initial *trigger cubes* for each required cube separately. A trigger cube is taken as an initial cover, and it includes signals that must appear in any correct cover of the corresponding required cube. Since this initial cover may include minterms from the OFF-set, called *cover violations* (CV), as well as dynamic hazards, called *intersection violations* (IV), additional *context signals* may have to be added to the cover. A context signal is any signal that is stable in the required cube and which is not a trigger signal. Next, a binate covering problem is formulated to remove all cover and intersection violations by selecting the minimum number of context signals needed. The result is a set of solutions derived for each required cube. Finally, the results from each of the subproblems are merged together by setting up and solving a unate covering problem for each output signal. The goal of this covering problem is to find the minimum number of the solutions found for the required cubes for this signal that are needed to cover all required cubes for this output signal.

*Algorithm VI.2:*

```

required_cubes( $\langle I, O, \Phi, \Gamma \rangle, u$ )
  RC =  $\emptyset$ ;
  foreach  $s \in \Phi$ 
    if  $s(u) = R$  then // Dynamic  $0 \rightarrow 1$  and static  $1 \rightarrow 1$ 
      RC = RC  $\cup$  cube(s);
    else if  $s(u) = 1$  then
      if  $\exists (s, s') \in \Gamma . s'(u) = 1$  then // Static  $1 \rightarrow 1$ 
        RC = RC  $\cup$  cube(s);
      else // Dynamic  $1 \rightarrow 0$  transition
        foreach  $v \in I \cup O$ 
          if ( $s(v) = R$  and  $s'(v) = 1$ ) or
             ( $s(v) = F$  and  $s'(v) = 0$ ) then
            RC = RC  $\cup$  split(s, v);
        else if  $s(u) = 0$  or  $s(u) = F$  then ; // No required cube
  return(RC);

```

Fig. 6. Algorithm to find required cubes.

### A. Finding required cubes

The set of required cubes describes how the ON-set of the function must be covered in order to obtain a hazard-free solution. Each required cube must be completely covered by some product term. The following required cubes must be generated for each generalized transition  $[c_1, c_2]$ :

- $0 \rightarrow 1$  transition: required cube equals  $c_2$ .
- $1 \rightarrow 1$  transition: required cube equals  $[c_1, c_2]$ .
- $1 \rightarrow 0$  transition: set of required cubes is the maximal subcubes of  $[c_1, c_2]$ .
- $0 \rightarrow 0$  transition: no required cube.

Required cubes are derived during a traversal of the CSG as shown in Figure 6. For each compacted state,  $s$ , the algorithm checks the value of the output signal,  $u$ , that is being synthesized. If the value of  $u$  in  $s$  is  $R$ , then the compacted state is an output burst state representing a  $0 \rightarrow 1$  transition. This has a required cube equal to the initial cube of this burst. This state also represents a  $1 \rightarrow 1$  transition when the output signals are fed back. This cube is equal to the entire state cube which includes the first required cube. Therefore, only the second required cube is necessary.

If the value of  $u$  in  $s$  is 1, this represents either a  $1 \rightarrow 1$  transition or a  $1 \rightarrow 0$  transition. If it is a  $1 \rightarrow 1$  transition then the successor state also has  $u$  as 1. In this case, the required cube is the state cube. If the value of the output in the successor state is  $F$ , then the required cubes are all the maximal subcubes of  $s$ . These can be found by checking each unstable signal for stability in the successor state. If this signal is stabilizing, then a cube is formed where all other unstable signals are don't care, and this signal is stable at its initial value.

If the value of  $u$  in  $s$  is 0, this represents either a  $0 \rightarrow 0$  transition or a  $0 \rightarrow 1$  transition. A required cube is only needed if it is a  $0 \rightarrow 1$  transition. As mentioned above, this required cube is included in the next compacted state (the output burst state). Therefore, it does not need to be generated for this compacted state.

If the value of  $u$  is  $F$ , then the compacted state must be an output burst state representing a  $1 \rightarrow 0$  transition. The required cubes for this transition are found when the preceding compacted state (the input burst) is processed

as described above.

Consider the CSG in Figure 4 as an example. The notation  $x+,3$  corresponds to the third occurrence of signal  $x$  rising in the CSG in a depth first search (left branch first) starting in the start state  $abdl.xyzuv:0RR-.00001$ . Similarly,  $x*,1$  refers to the first static transition of  $x$ , and  $x-,1$  to the first falling transition of  $x$ . Consider the output signal  $x$ . The logic minimization starts out by first deriving the required cubes for  $x$ . Take required cube  $RC(x+,2) = 110-.0-01$  as an example. This required cube corresponds to the leftmost shaded compacted state,  $S = 110-.R0R01$ , in Figure 4 where  $x$  is enabled to rise. This required cube covers the static  $1 \rightarrow 1$  transition for  $x$  caused by the output variable feedback in state  $S$ . The required cubes for  $x$  are illustrated in Table II.

### B. Finding trigger cubes and context signals

Each required cube has an associated trigger cube which forms the initial cover for that required cube. While a final hazard-free cover cube can always be found from an empty trigger cube, adding initial trigger signals narrows the search for the cover cube. These trigger signals are formed by signals known to be required for a legal cover and are derived from the current transition.

The transition cube of a  $0 \rightarrow 1$  transition  $[c_1, c_2]$  is restricted to turn on only once the end cube  $c_2$  has been reached.  $c_2$  is reached only once all terminating signals in the transition cube have fired. The final cover cube for this transition is therefore required to contain the value of all terminating edge signals after they have fired — otherwise the final cover cube would intersect the OFF-set. The terminating edge signals of the transition are therefore added to the trigger cube as initial trigger signals.

Similarly, each trigger cube for a  $1 \rightarrow 0$  transition  $[c_1, c_2]$  is required to turn off as the end cube  $c_2$  is entered. Each required cube, being a maximal subcube of  $[c_1, c_2]$ , must therefore turn off as the last signal in the trajectory it covers fires. Each trigger cube can therefore be annotated with the initial value of the last signal to fire in its corresponding required cube.

Note that a signal specified as a directed don't care in a transition cannot be a trigger signal since that transition has its value as a don't care throughout the transition cube. A level signal is also not a trigger signal since the setup time requirement forces it to stabilize before any compulsory edges arrive at the inputs. Once a trigger cube is found, the context signals are those signals that are not trigger signals and are stable in the required cube.

In our example, after generating the required cubes, the algorithm proceeds to find the context signals and trigger cubes corresponding to these required cubes. For the required cube  $RC(x+,2)$  derived from state  $110-.R0R01$  in our example, the corresponding trigger cube is  $TC(x+,2) = ----.----1$  since  $v+$  is the signal transition that causes the machine to enter the required cube and turn on the cover. The corresponding context signals are  $a, b, d', y', u'$  since these signals are stable throughout the required cube. The trigger cubes and context signals

TABLE II

REQUIRED CUBES, TRIGGER CUBES, AND CONTEXT SIGNALS FOR THE TRANSITIONS OF OUTPUT  $x$  OF CONTROLLER  $ack-xbm-si$ .

|         | Required cubes | Trigger cubes | Context signals     |
|---------|----------------|---------------|---------------------|
| $x+,1$  | 01--.-000-     | -1--.-----    | $a' y' z' u'$       |
| $x*,1$  | -1--.10000     | -----         | $b x y' z' u' v'$   |
| $x-,1$  | 111-.10000     | -1--.-----    | $a d x y' z' u' v'$ |
| $x+,2$  | 110-.0-01      | -----1        | $a b d' y' u'$      |
| $x*,2$  | 11--.10101     | -----         | $a b x y' z u' v$   |
| $x*,3$  | 111-.10-0-     | -----         | $a b d x y' u'$     |
| $x+,3$  | 110--0-01      | -----1-       | $a b d' z' v'$      |
| $x*,4$  | -1--.11010     | -----         | $b x y z' u v'$     |
| $x*,5$  | 01--.1-0-0     | -----         | $a' b x z' v'$      |
| $x-,2b$ | 01--.10000     | -1--.-----    | $a' x y' z' u' v'$  |
| $x-,2d$ | 0-0-.10000     | --0-.-----    | $a' x y' z' u' v'$  |

for  $x$  are illustrated in Table II.

### C. Finding violations

Since the initial trigger cube may span over minterms belonging to the OFF-set (i.e., a *cover violation*), and may also intersect other transition cubes in a hazardous way (i.e., an *intersection violation*), such violations must be removed.

A trigger cube  $tc$  for a signal  $u$  contains a cover violation if it intersects a transition  $[c_1, c_2]$  in which the value of  $u$  is 0. The algorithm shown in Figure 7 finds all cover violations. In a compacted state  $s$ , signal  $u$  is 0 when it has value 0, and it is tending to 0 when it has value  $F$ . When signal  $u$  is 0 or tending to 0 and if  $tc$  intersects  $s$ , then there may be a cover violation. There is, however, one special case. If the value of  $u$  is 0 in an input burst and  $R$  in the following output burst then all states in the compacted state except the final state (where  $u$  is tending to 1) must be excluded. Therefore, in this case all maximal subcubes must be found and excluded individually (if they intersect the trigger cube).

#### Algorithm VI.3:

```

cover_violations( $\langle I, O, \Phi, \Gamma \rangle, u, tc$ )
   $CV = \emptyset$ ;
  foreach  $s \in \Phi$ 
    if ( $s(u) = 0$  or  $s(u) = F$ ) and  $tc \cap s \neq \emptyset$  then
      if  $\exists (s, s') \in \Gamma . s'(u) = R$  then
        foreach  $v \in I \cup O$ 
          if ( $(s(v) = R$  and  $s'(v) = 1)$  or
              ( $s(v) = F$  and  $s'(v) = 0$ )) and
              ( $\text{split}(s, v) \cap tc \neq \emptyset$ ) then
             $CV = CV \cup \text{split}(s, v)$ ;
          else
             $CV = CV \cup s$ ;
  return( $CV$ );

```

Fig. 7. Algorithm to find cover violations.

A trigger cube  $tc$  contains an intersection violation (also called illegal intersection) if  $tc$  intersects a dynamic  $1 \rightarrow 0$  transition  $[c_1, c_2]$  and  $tc$  does not contain the start subcube  $c'_1$ . An intersection violation also exists if  $tc$  intersects the end cube,  $c_2$  for a dynamic  $0 \rightarrow 1$  transition and does not contain the end subcube  $c'_2$ . Intersection violations can also be caused by the selection of a context signal that, when added to the trigger cube, causes an illegal intersection of a dynamic transition.

Intersection violations can be detected through a state graph traversal as shown in the algorithm in Figure 8. This algorithm returns a set of intersection violations in which each violation is a pair where the first element is the choice of context signal that causes the violations and the second is the state which must be excluded.

*Algorithm VI.4:*

```

intersection_violations( $(I, O, \Phi, \Gamma), CS, u, rc, tc$ )
   $IV = \emptyset$ ;
  foreach  $s \in \Phi$ 
    // Check if is dynamic 1  $\rightarrow$  0 intersecting trigger cube
    if  $s(u) = 1$  and  $\exists (s, s') \in \Gamma . s'(u) = F$  and  $tc \cap s \neq \emptyset$  then
       $c'_1 = \text{initial}(s)$ ; // Find start subcube
      if  $c'_1 \not\subseteq tc$  then
         $IV = IV \cup \{ (\emptyset, s) \}$ ;
      else
        foreach  $v \in CS$ 
          // Check if context signal excludes part of
          // start subcube but not the state
          if  $rc(v) \neq c'_1(v)$  and  $\text{cube}(s)(v) = -$  then
             $IV = IV \cup \{ (v, s) \}$ ;
        // Check if state is dynamic 0  $\rightarrow$  1 intersecting trigger cube
        else if  $s(u) = R$  and  $tc \cap s \neq \emptyset$  then
          // Continue if there are level signals or directed don't cares
          if  $\exists v \in I . \text{cube}(s)(v) = -$  then
             $c'_2 = s$ ; // Copy for end subcube
            foreach  $v \in O$ 
               $s'(v) = \text{initial}(s)(v)$ ;
               $c'_2 = \text{final}(s')$ ; // Find end subcube
            if  $tc \cap s'$  then
              if  $c'_2 \not\subseteq tc$  then
                 $IV = IV \cup \{ (\emptyset, s') \}$ ;
              else
                foreach  $v \in CS$ 
                  // Check if context signal excludes part
                  // of end subcube but not the state
                  if  $rc(v) \neq c'_2(v)$  and  $\text{cube}(s')(v) = -$  then
                     $IV = IV \cup \{ (v, s') \}$ ;
          return( $IV$ );

```

Fig. 8. Algorithm to find intersection violations.

For each dynamic 1  $\rightarrow$  0 transition state  $s$ , the algorithm checks if the trigger cube,  $tc$ , intersects  $s$ , but does not contain the start subcube. The start subcube,  $c'_1$ , is found by setting all changing signals to their initial value. Even if  $tc$  contains  $c'_1$ , the choice of a context signal may lead to an illegal intersection, and a context signal must be added to remove it. In other words, if a choice of a context signal causes the cover to exclude part of the start subcube but still intersect the state, then we must choose another context signal that excludes the whole state  $s$ .

For each dynamic 0  $\rightarrow$  1 transition, if there are no unstable input signals or conditional signals, then the end cube equals the end subcube and is a single minterm so there can be no violation (i.e., this is a burst-mode transition). If this is not the case, we set all outputs in the state to their initial value since the 0  $\rightarrow$  1 transition takes place before the changes in output values are allowed to be fed back. To calculate the end subcube, we set all unstable input signals to their final value. If the trigger cube intersects this compacted state, but does not include the end subcube, then there is an intersection violation and a context signal must be added to remove this compacted state. If a context signal choice can exclude part of the end sub-

TABLE III

COVER VIOLATIONS (CV) AND INTERSECTION VIOLATIONS (IV) FOR THE TRIGGER CUBES (TC) OF OUTPUT  $x$  OF CONTROLLER *ack-xbm-si*.

| TC       | CV                                                                                                                             | IV         |
|----------|--------------------------------------------------------------------------------------------------------------------------------|------------|
| $x+, 1$  | 1101.00000, 11F1.00000<br>1100.00000, 11F0.00000                                                                               | none       |
| $x*, 1$  | 1R1-.00000, 1101.00000<br>11F1.00000, 1100.00000<br>11F0.00000, 101-.F0000<br>00R-.00001, 001-.F0000<br>000-.0000R, 00F-.00000 | none       |
| $x-, 1$  | CV( $x+, 1$ )                                                                                                                  | 0FR-.10000 |
| $x+, 2$  | 00R-.00001                                                                                                                     | 01R-.00001 |
| $x*, 2$  | CV( $x*, 1$ )                                                                                                                  | none       |
| $x*, 3$  | CV( $x*, 1$ )                                                                                                                  | 0FR-.10000 |
| $x+, 3$  | none                                                                                                                           | none       |
| $x*, 4$  | CV( $x*, 1$ )                                                                                                                  | none       |
| $x*, 5$  | CV( $x*, 1$ )                                                                                                                  | none       |
| $x-, 2b$ | CV( $x+, 1$ )                                                                                                                  | none       |
| $x-, 2d$ | 1101.00000, 1100.00000<br>00R-.00001, 000-.0000R                                                                               | 01R-.00001 |

cube while still allowing the cover to intersect the state, this creates a new intersection violation. This intersection violation must be removed by choosing additional context signals that remove the rest of the state.

Consider trigger cube  $TC(x+, 2) = \text{----.----}1$  in our example. This trigger cube has a cover violation since it intersects compacted state  $0RR-.00001$  which belongs to the OFF-set of  $x$ . Since  $x$  goes high once  $b$  rises, the cover violation is represented by cube  $00R-.00001$ . The trigger cube also has a potential intersection violation of state  $0RR-.00001$ . If  $d'$  was to be selected as a context signal, the trigger cube would no longer cover the end subcube  $011-.00001$  of the transition and the cover would be hazardous. If this context signal is added to the cover (to remove other violations), the entire end cube  $01R-.00001$  of state  $0RR-.00001$  must be excluded in order to remove the introduced hazard. This is achieved by selecting other context signals that exclude the entire end cube from the cover. The cover and intersection violations for output  $x$  are shown in Table III.

*D. Finding covers*

In order to find the best choice of context signals to remove all violations, our logic minimization algorithm sets up and solves a covering problem where the columns of the covering table are the available context signals and the rows are the violations that must be removed. Since selecting certain context signals can lead to more intersection violations, this covering problem is binate. A row is added to the binate cover table for every cover violation along with an "X" in the column for every context signal that completely removes the violating state cube. Similarly, a row is added to the binate table for every intersection violation. An "X" is added for every context signal that removes the violation. In addition an "O" is added if the intersection violation is caused by that context signal.

The binate cover table is then solved using classic reduction techniques and a branch and bound algorithm. During column dominance, a dominated context signal is only removed if the dominating context signal excludes less required cubes. Since our ultimate goal is to find the mini-

| $x^*,1$    | b x y' z' u' v' | $x^*,2$    | a b x y' z u' v |
|------------|-----------------|------------|-----------------|
| 1R1-.00000 | X               | 1R1-.00000 | X X X           |
| 1101.00000 | X               | 1101.00000 | X X X           |
| 11F1.00000 | X               | 11F1.00000 | X X X           |
| 1100.00000 | X               | 1100.00000 | X X X           |
| 11F0.00000 | X               | 11F0.00000 | X X X           |
| 101-.F0000 | X               | 101-.F0000 | X X X           |
| 00R-.00001 | X X X           | 00R-.00001 | X X X X         |
| 001-.F0000 | X               | 001-.F0000 | X X X X         |
| 000-.0000R | X X             | 000-.0000R | X X X X         |
| 00F-.00000 | X X             | 00F-.00000 | X X X X         |

| $x^*,3$    | a b d x y' u' | $x^*,4$    | b x y z' u' v' |
|------------|---------------|------------|----------------|
| 1R1-.00000 | X             | 1R1-.00000 | X X X          |
| 1101.00000 | X X           | 1101.00000 | X X X          |
| 11F1.00000 | X             | 11F1.00000 | X X X          |
| 1100.00000 | X X           | 1100.00000 | X X X          |
| 11F0.00000 | X             | 11F0.00000 | X X X          |
| 101-.F0000 | X             | 101-.F0000 | X X X          |
| 00R-.00001 | X X X         | 00R-.00001 | X X X X        |
| 001-.F0000 | X X           | 001-.F0000 | X X X          |
| 000-.0000R | X X X X       | 000-.0000R | X X X X        |
| 00F-.00000 | X X X         | 00F-.00000 | X X X X        |
| 0FR-.10000 | X O           |            |                |

| $x^*,5$    | a' b x z' v' | $x^*,1$    | a d x y' z' u' v' |
|------------|--------------|------------|-------------------|
| 1R1-.00000 | X X          | 11F1.00000 | X                 |
| 1101.00000 | X X          | 1101.00000 | X X               |
| 11F1.00000 | X X          | 11F0.00000 | X                 |
| 1100.00000 | X X          | 1100.00000 | X X               |
| 11F0.00000 | X X          | 0FR-.10000 | X O               |
| 101-.F0000 | X X          |            |                   |
| 00R-.00001 | X X X        |            |                   |
| 001-.F0000 | X            |            |                   |
| 000-.0000R | X X          |            |                   |
| 00F-.00000 | X X          |            |                   |

| $x+,1$     | a' y' z' u' | $x-,2b$    | a' x y' z' u' v' |
|------------|-------------|------------|------------------|
| 11F1.00000 | X           | 11F1.00000 | X X              |
| 1101.00000 | X           | 1101.00000 | X X              |
| 11F0.00000 | X           | 11F0.00000 | X X              |
| 1100.00000 | X           | 1100.00000 | X X              |

| $x+,2$     | a b d' y' u' | $x-,2d$    | a' x y' z' u' v' |
|------------|--------------|------------|------------------|
| 00R-.00001 | X X          | 00R-.00001 | X X              |
| 01R-.00001 | X O          | 1101.00000 | X X              |
|            |              | 1100.00000 | X X              |
|            |              | 000-.0000R | X                |
|            |              | 01R-.00001 | X X              |

Fig. 9. Binate cover tables for output  $x$  of controller  $ack-xbm-si$ .

imum cover of all required cubes, this step is necessary to ensure that a minimum literal solution can be found. During the branch and bound, the algorithm records the set of minimal unique hazard-free cubes that can cover the required cube. A minimal unique cube is a cube covering a set of required cubes not covered by any other cube of smaller cardinality.

Once a set of hazard-free minimal unique cubes have been derived for all required cubes of an output function, finding the minimal cover is posed as a unate covering problem. In the unate table, the rows represent the required cubes of the function and the columns represent the sets of minimal unique cubes. Solving the unate cover table using classic techniques then results in a final hazard-free cover that is minimal in number of literals.

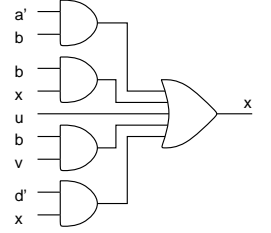
In our example, consider the binate cover table built for trigger cube  $TC(x+,2) = ----.----1$ . The context signals  $a, b, d', y', u'$  of the trigger cube are entered as columns

|                     |   |             |                     |   |           |
|---------------------|---|-------------|---------------------|---|-----------|
| LocalSol( $x+,1$ )  | = | a' b        | LocalSol( $x+,3$ )  | = | u         |
| LocalSol( $x^*,1$ ) | = | b x         | LocalSol( $x^*,4$ ) | = | u, b x    |
| LocalSol( $x-,1$ )  | = | b x         | LocalSol( $x^*,5$ ) | = | a' b, b x |
| LocalSol( $x+,2$ )  | = | b v         | LocalSol( $x-,2b$ ) | = | a' b, b x |
| LocalSol( $x^*,2$ ) | = | z, b v, b x | LocalSol( $x-,2d$ ) | = | d' x      |
| LocalSol( $x^*,3$ ) | = | b x         |                     |   |           |

a) Local solutions

| x            | a'b | bx | z | bv | u | d'x |
|--------------|-----|----|---|----|---|-----|
| 01--.-000-   | X   |    |   |    |   |     |
| -1--.-10000  |     | X  |   |    |   |     |
| 111-.10000   |     | X  |   |    |   |     |
| 110--.-0-01  |     |    |   | X  |   |     |
| 11--.-10101  |     | X  | X | X  |   |     |
| 111-.10-0-   |     | X  |   |    |   |     |
| 110--.-010   |     |    |   |    | X |     |
| -1--.-11010  |     | X  |   |    | X |     |
| 01--.-1-0-0  | X   | X  |   |    |   |     |
| 01--.-10000  | X   | X  |   |    |   |     |
| 0-0--.-10000 |     |    |   |    |   | X   |

b) Unate cover table



c) Final cover

Fig. 10. Local solutions, unate cover table, and final solution for output  $x$  of controller  $ack-xbm-si$ .

in the table and the two violation cubes  $00R-.00001$ , and  $01R-.00001$  are entered as rows. An “O” is entered for row  $01R-.00001$  in the column for  $d'$  since adding that context signal will introduce a hazard and require us to remove the row from the table by selecting another context signal. For each row an “X” is entered in each column where the context signal would remove the violation cube. In our case, signals  $a$  and  $b$  would remove cover violation  $00R-.00001$ , and  $a$  would remove intersection violation  $01R-.00001$ . In this fashion, binate cover tables are built for each required cube for  $x$  as illustrated in Figure 9.

The binate cover tables are then solved one by one. For trigger cube  $x+,2$  there exists two minimal unique solution cubes,  $bv$  and  $av$ . Since they both cover the same set of required cubes with the same number of literals, only one cube needs to be kept. In our case we keep  $bv$ . Generally, a trigger cube can have several local solutions.  $x^*,2$  for example has three different local solutions,  $z, bv$ , and  $bx$ . These are all minimal unique solutions that cover a set of required cubes not covered by any of the other local solutions. For example, while  $z$  is a smaller solution than  $bv$ , it covers only required cube  $11--.10101$  while  $bv$  covers  $11--.10101$  and  $110--.-0-01$ . Thus  $bv$  is a minimal solution for the unique set of required cubes it covers. Similarly, while  $z$  covers only a subset of the required cubes that  $bv$  covers,  $z$  is a smaller cover, so  $z$  is still a minimal cover for the set of required cubes  $\{11--.10101\}$ . The local solutions for each trigger cube are illustrated in Figure 10(a).

Finally, a unate cover table (see Figure 10(b)) is built with the required cubes as rows, and the combined local solutions as columns. A final minimal literal solution covering all required cubes is then found using classic reduction techniques. In our case, local solution cubes  $ab', bx, bv, u$ , and  $d'x$  are all essential and form the final cover illustrated in Figure 10(c).

*Algorithm VII.1:*

```

minimize( $\langle I, O, \Phi, \Gamma \rangle$ )
  foreach  $u \in O$ 
    foreach  $p \in \{set, reset\}$ 
      LocalSol =  $\emptyset$ ;
      RC = required_cubes( $\langle I, O, \Phi, \Gamma \rangle, u, p$ );
      foreach  $rc \in RC$ 
        tc = trigger_cube( $\langle I, O, \Phi, \Gamma \rangle, u, p$ );
        CS = context_signals( $I, O, rc, tc$ );
        CV = cover_violations( $\langle I, O, \Phi, \Gamma \rangle, u, p, tc$ );
        IV = intersection_violations( $\langle I, O, \Phi, \Gamma \rangle, CS, u, p, rc, tc$ );
        table = build_binate_table(tc, CS, CV, IV);
        LocalSol = LocalSol  $\cup$  solve_binate_table(table, RC);
      table = build_unate_table(RC, LocalSol);
      Sol( $u_p$ ) = solve_unate_table(table);
    return(Sol);

```

Fig. 11. Minimization algorithm for gC Circuits.

VII. GC MINIMIZATION ALGORITHM

The logic minimization algorithm for gC circuits is very similar to that for standard gate minimization. There are, however, two major differences. In a standard gate implementation, the circuit implements only one function, the ON-set function. A gC implementation, however, implements two, the set and the reset functions. Hazard-free logic must therefore be synthesized for both of these functions. In addition, because of the different hazard properties of gC circuits, some of the standard gate hazard constraints can be relaxed, potentially producing smaller covers.

Figure 11 presents our logic minimization algorithm for gC implementations. As in the standard gate case, the input to the logic minimization algorithm is a CSG derived from a state assigned XBM machine. The output is two minimum literal logic equations for each output signal implementing the set and reset regions in the CSG. As illustrated by the algorithm in Figure 11 the set and reset functions are treated separately during minimization of gC circuits. When minimizing the set function,  $f_{set}$ , the set regions form the ON-set of the function. The OFF-set is formed by the reset regions of the reset function and the CSG states in which the output is defined to be low. Similarly, when minimizing the reset function,  $f_{reset}$ , the reset regions form the ON-set. The OFF-set is formed by the set regions of the set function and the CSG states in which the output is defined to be high.

A. Finding required cubes

In the gC case, there are two sets of required cubes, one for the set function,  $f_{set}$ , and one for the reset function,  $f_{reset}$ . In contrast to the standard gate case, since gC circuits make use of a staticizer to hold the output stable between set and reset regions, there are no required cubes for static transitions. The following required cubes must be generated for each generalized transition  $[c_1, c_2]$ :

- $0 \rightarrow 1$  transition: required cube for  $f_{set}$  equals  $c_2$ .
- $1 \rightarrow 1$  transition: no required cube.
- $1 \rightarrow 0$  transition: required cube for  $f_{reset}$  equals  $c_2$ .
- $0 \rightarrow 0$  transition: no required cube.

*Algorithm VII.2:*

```

required_cubes( $\langle I, O, \Phi, \Gamma \rangle, u, p$ )
  RC =  $\emptyset$ ;
  foreach  $s \in \Phi$ 
    if ( $(s(u) = R)$  and ( $p = set$ )) then
      RC = RC  $\cup$  cube( $s$ );
    else if ( $(s(u) = F)$  and ( $p = reset$ )) then
      RC = RC  $\cup$  cube( $s$ );
  return(RC);

```

Fig. 12. Algorithm to find required cubes for gC.

The required cubes are derived during a traversal of the CSG as shown in Figure 12. For each compacted state,  $s$ , the algorithm checks the value of the output signal,  $u$ , that is being synthesized. If the value of  $u$  in  $s$  is  $R$ , then the compacted state is an output burst state representing a  $0 \rightarrow 1$  transition. If required cubes are currently being generated for the set function, this transition has a required cube equal to the final cube of the transition. If the value of  $u$  in  $s$  is 1 or 0, then the compacted state is a static transition. Static transitions do not have required cubes in a gC circuit. If the value of  $u$  is  $F$ , then the compacted state must be an output burst state representing a  $1 \rightarrow 0$  transition. If required cubes are currently being generated for the reset function, this transition has a required cube equal to the final cube of the transition. The gC required cubes for output  $x$  for the example in Figure 4 are shown in Table IV.

B. Finding trigger cubes and context signals

As in the standard gate case, the cover cube of a  $0 \rightarrow 1$  transition  $[c_1, c_2]$  is restricted to turn on only once the end cube  $c_2$  has been reached.  $c_2$  is reached only once all terminating signals in the transition cube have fired. When the set function is generated, the terminating edge signals at their final value are added as trigger signals to the trigger cube for this transition. The same argument applies to trigger cubes for the reset function when  $1 \rightarrow 0$  transitions are considered. The gC trigger cubes for  $x$  are shown in Table IV along with their respective context signals.

TABLE IV  
REQUIRED CUBES, TRIGGER CUBES, AND CONTEXT SIGNALS FOR THE TRANSITIONS OF OUTPUT  $x$  OF CONTROLLER *ack-xbm-si*.

|                | Required cubes | Trigger cubes | Context signals    |
|----------------|----------------|---------------|--------------------|
| $f_{set}(x)$   |                |               |                    |
| $x+,1$         | 01-- .00001    | -1-- .-----   | a' x' y' z' u' v   |
| $x+,2$         | 110- .00001    | ---- .----1   | a b d' x' y' z' u' |
| $x+,3$         | 110- .00010    | ---- .---1-   | a b d' x' y' z' v' |
| $f_{reset}(x)$ |                |               |                    |
| $x-,1$         | 101- .10000    | -0-- .-----   | a d x y' z' u' v'  |
| $x-,2$         | 001- .10000    | -01- .-----   | a' x y' z' u' v'   |

C. Finding violations and covers

As in the standard gate case, an initial trigger cube may span over minterms belonging to the OFF-set (i.e., a *cover violation*), and may also intersect other transition cubes in a hazardous way (i.e., an *intersection violation*). Such violations must be removed.

The way to detect cover violations is the same as for the standard gate case. However, checks must be made

**Algorithm VII.3:**

```

cover_violations( $(I, O, \Phi, \Gamma), u, p, tc$ )
   $CV = \emptyset$ ;
  foreach  $s \in \Phi$ 
    if ( $(p = set)$  and  $(s(u) = 0$  or  $s(u) = F)$ ) or
       ( $p = reset$  and  $(s(u) = 1$  or  $s(u) = R)$ )) and
        $tc \cap s \neq \emptyset$  then
      if ( $\exists (s, s') \in \Gamma . s(u) = 0$  and  $s'(u) = R$ ) or
          ( $\exists (s, s') \in \Gamma . s(u) = 1$  and  $s'(u) = F$ ) then
        foreach  $v \in I \cup O$ 
          if ( $(s(v) = R$  and  $s'(v) = 1)$  or
              ( $s(v) = F$  and  $s'(v) = 0$ )) and
              ( $split(s, v) \cap tc \neq \emptyset$ ) then
             $CV = CV \cup split(s, v)$ ;
          else
             $CV = CV \cup s$ ;
  return( $CV$ );

```

Fig. 13. Algorithm to find cover violations for gC.

**Algorithm VII.4:**

```

intersection_violations( $(I, O, \Phi, \Gamma), CS, u, p, rc, tc$ )
   $IV = \emptyset$ ;
  foreach  $s \in \Phi$ 
    if ( $p = set$  and  $s(u) = R$ ) or
       ( $p = reset$  and  $s(u) = F$ ) and  $tc \cap s \neq \emptyset$  then
      // Continue if level signals or directed don't cares
      if  $\exists v \in I . cube(s)(v) = -$  then
         $c'_2 = s$ ; // Copy for end subcube
        foreach  $v \in O$ 
           $s'(v) = initial(s)(v)$ ;
           $c'_2 = final(s')$ ; // Find end subcube
          if  $tc \cap s'$  then
            if  $c'_2 \not\subseteq tc$  then
               $IV = IV \cup \{ (\emptyset, s') \}$ ;
            else
              foreach  $v \in CS$ 
                // Check if context signal excludes part
                // of end subcube but not the state
                if  $rc(v) \neq c'_2(v)$  and  $cube(s')(v) = -$  then
                   $IV = IV \cup \{ (v, s') \}$ ;
  return( $IV$ );

```

Fig. 14. Algorithm to find intersection violations for gC.

separately for the set and reset functions as the set function is concerned with intersecting the OFF-set while the reset function is concerned with intersecting the ON-set.

A gC trigger cube  $tc$  belonging to the set function contains an intersection violation if  $tc$  intersects the end cube,  $c_2$  of a dynamic  $0 \rightarrow 1$  transition and does not contain the end subcube  $c'_2$ . Similarly, a trigger cube  $tc$  belonging to the reset function contains an intersection violation if  $tc$  intersects the end cube,  $c_2$  of a dynamic  $1 \rightarrow 0$  transition and does not contain the end subcube  $c'_2$ . In contrast to the standard gate case, there are no intersection violations due to trigger cubes intersecting the start cube of a transition without containing the start subcube. Since transistor stacks within a gC gate are assumed to switch simultaneously when triggered by the same signal, and the decomposition of such stacks are restricted [10], hazards cannot occur due to such intersections. As in the standard gate case, intersection violations can be detected through a state graph traversal as shown in Figure 14.

As in the standard gate case, when considering the gC set function, for each dynamic  $0 \rightarrow 1$  transition, if there are no unstable input signals or conditional signals, there can

TABLE V

COVER VIOLATIONS (CV) AND INTERSECTION VIOLATIONS (IV) FOR THE TRIGGER CUBES (TC) OF OUTPUT  $x$  OF CONTROLLER *ack-xbm-si*.

| TC             | CV                                             | IV         |
|----------------|------------------------------------------------|------------|
| $f_{set}(x)$   |                                                |            |
| $x+,1$         | 11F1.00000,1101.00000<br>11F0.00000,1100.00000 | none       |
| $x+,2$         | 00R-.00001                                     | 01R-.00001 |
| $x+,3$         | none                                           | none       |
| $f_{reset}(x)$ |                                                |            |
| $x-,1$         | 0F0-.10000                                     | none       |
| $x-,2$         | none                                           | none       |

| $x+,1$     | a' x' y' z' u' v | $x+$       | bv u |
|------------|------------------|------------|------|
| 11F1.00000 | X X X            | 01--.00001 | X    |
| 1101.00000 | X X X            | 110-.00001 | X    |
| 11F0.00000 | X X X            | 110-.00010 | X    |
| 1100.00000 | X X X            |            |      |

| $x+,2$     | a b d' x' y' z' u' | $x-$       | b'd |
|------------|--------------------|------------|-----|
| 01R-.00001 | X 0                | 101-.10000 | X   |
| 00R-.00001 | X X                | 001-.10000 | X   |

| $x-,1$     | a d x y' z' u' v' |
|------------|-------------------|
| 0F0-.10000 | X X               |

a) Binate cover tables

LocalSol( $x+,1$ ) = b v  
 LocalSol( $x+,2$ ) = b v  
 LocalSol( $x+,3$ ) = u  
 LocalSol( $x-,1$ ) = b' d  
 LocalSol( $x-,2$ ) = b' d

b) Local solutions

c) Unate cover tables

d) Final cover

Fig. 15. Binate cover tables, local solutions, unate cover table, and final solution for output  $x$  of controller *ack-xbm-si*.

be no violation. If this is not the case, all outputs in the state are set to their initial value since the  $0 \rightarrow 1$  transition takes place before the changes in output values are allowed to be fed back. To calculate the end subcube, all unstable input signals are set to their final value. If the trigger cube intersects this compacted state, but does not include the end subcube, then there is an intersection violation and a context signal must be added to remove this compacted state. If a context signal choice can exclude part of the end subcube while still allowing the cover to intersect the state, this creates a new intersection violation which must be removed by choosing additional context signals that remove the rest of the state. The same arguments apply to the reset function when considering dynamic  $1 \rightarrow 0$  transitions. The cover and intersection violations for output  $x$  of controller *ack-xbm-si* are shown in Table V. The cover tables and final solution are illustrated in Figure 15.

## VIII. RESULTS

The algorithm described in this article has been completely incorporated and automated in the ATACS [21] synthesis tool. The ATACS extended burst-mode logic minimizer is exact in number of literals. We compare against the publicly available state of the art hazard-free logic minimization tools developed by Nowick et al. The HFMIN [6] minimizer is exact in number of literals. The IMPYMIN

TABLE VI

Benchmark comparisons of gC implementations between existing two-level logic minimizers ATACS (method presented in this article), HFMIN, IMPYMIN, and ESPRESSO-HF. ( $|\Phi|$  - number of compacted states,  $IO$  - number of input, output, and state signals,  $Time$  - minimization run-time in seconds,  $Lit$  - literals in solution.)

| Design            | $ \Phi $ | IO | ATACS<br>(literal-exact) |     | HFMIN<br>(literal-exact) |     | IMPYMIN<br>(cube-exact) |     | ESPRESSO-HF<br>(heuristic) |     |
|-------------------|----------|----|--------------------------|-----|--------------------------|-----|-------------------------|-----|----------------------------|-----|
|                   |          |    | Time                     | Lit | Time                     | Lit | Time                    | Lit | Time                       | Lit |
| ack-cdplayer-p1   | 74       | 45 | 0.42                     | 274 | impossible               |     | 150.86                  | 276 | 20.02                      | 277 |
| ack-fibonacci     | 50       | 35 | 0.34                     | 194 | 1675.65                  | 194 | 85.74                   | 204 | 13.61                      | 198 |
| ack-diffeq        | 32       | 34 | 0.33                     | 177 | 120.16                   | 177 | 45.93                   | 184 | 14.32                      | 185 |
| ack-barcode       | 34       | 28 | 0.31                     | 167 | 45.90                    | 167 | 31.54                   | 168 | 11.15                      | 168 |
| ack-gcd           | 34       | 20 | 0.33                     | 80  | 19.26                    | 80  | 16.44                   | 80  | 6.48                       | 80  |
| ack-factorial     | 23       | 17 | 0.27                     | 42  | 16.01                    | 42  | 13.26                   | 42  | 5.77                       | 42  |
| cache-ctrl        | 98       | 36 | 0.79                     | 510 | 2524.96                  | 510 | 273.40                  | 534 | 14.62                      | 521 |
| chu-ad-opt-e      | 8        | 6  | 0.25                     | 12  | 5.53                     | 12  | 4.68                    | 12  | 2.19                       | 12  |
| dme-e             | 20       | 8  | 0.28                     | 22  | 9.33                     | 22  | 8.50                    | 22  | 3.69                       | 22  |
| dme-fast-e        | 20       | 8  | 0.26                     | 27  | 9.27                     | 27  | 7.74                    | 29  | 3.66                       | 28  |
| dram-ctrl         | 25       | 13 | 0.31                     | 38  | 12.93                    | 38  | 10.25                   | 42  | 4.37                       | 42  |
| hp-ir-sc-ctrl     | 76       | 30 | 0.39                     | 205 | 81.73                    | 205 | 41.90                   | 216 | 12.26                      | 213 |
| hp-ir-sd-ctrl     | 54       | 24 | 0.50                     | 137 | 39.42                    | 137 | 30.24                   | 144 | 11.28                      | 144 |
| hp-ir-it-ctrl     | 23       | 13 | 0.26                     | 46  | 15.25                    | 46  | 12.85                   | 48  | 5.73                       | 47  |
| hp-ir-rf-ctrl     | 26       | 13 | 0.31                     | 33  | 13.18                    | 33  | 10.92                   | 35  | 5.06                       | 35  |
| hp-ir-two-tick    | 15       | 6  | 0.25                     | 8   | 3.82                     | 8   | 3.10                    | 8   | 1.59                       | 8   |
| hp-ir             | 16       | 5  | 0.26                     | 11  | 3.74                     | 11  | 3.09                    | 11  | 1.53                       | 11  |
| postoffice-pesnd  | 28       | 10 | 0.31                     | 57  | 9.33                     | 57  | 8.01                    | 58  | 3.64                       | 58  |
| postoffice-bufsnd | 17       | 8  | 0.33                     | 30  | 9.22                     | 30  | 7.61                    | 31  | 3.55                       | 31  |
| postoffice-bufrd  | 14       | 7  | 0.26                     | 15  | 7.32                     | 15  | 6.03                    | 15  | 2.88                       | 15  |
| pscsi-pscsi       | 119      | 20 | 0.58                     | 270 | 30.76                    | 270 | 22.49                   | 282 | 7.42                       | 280 |
| pscsi-isend       | 22       | 10 | 0.31                     | 48  | 11.09                    | 48  | 9.24                    | 48  | 4.25                       | 48  |
| pscsi-irvc        | 13       | 9  | 0.32                     | 26  | 9.15                     | 26  | 7.56                    | 26  | 3.56                       | 26  |
| pscsi-trcv-bm     | 18       | 10 | 0.27                     | 30  | 11.06                    | 30  | 10.01                   | 30  | 4.26                       | 30  |
| pscsi-trcv        | 14       | 8  | 0.32                     | 23  | 7.33                     | 23  | 6.09                    | 23  | 2.83                       | 23  |
| pscsi-tsend-bm    | 24       | 10 | 0.33                     | 45  | 11.26                    | 45  | 9.22                    | 45  | 4.21                       | 45  |
| pscsi-tsend       | 24       | 10 | 0.32                     | 50  | 11.14                    | 50  | 9.23                    | 50  | 4.26                       | 50  |
| sscsi-isend-bm    | 24       | 11 | 0.31                     | 52  | 11.39                    | 52  | 9.35                    | 52  | 4.24                       | 52  |
| sscsi-isend-csm   | 18       | 11 | 0.28                     | 39  | 11.05                    | 39  | 9.28                    | 41  | 4.31                       | 39  |
| sscsi-trcv-bm     | 24       | 11 | 0.27                     | 50  | 11.14                    | 50  | 9.39                    | 51  | 4.25                       | 51  |
| sscsi-trcv-csm    | 18       | 11 | 0.26                     | 37  | 11.04                    | 37  | 9.28                    | 39  | 4.23                       | 37  |
| sscsi-tsend-bm    | 26       | 11 | 0.30                     | 52  | 11.92                    | 52  | 9.32                    | 52  | 4.27                       | 52  |
| sscsi-tsend-csm   | 22       | 11 | 0.30                     | 38  | 11.10                    | 38  | 9.27                    | 39  | 4.22                       | 40  |
| stetson-p1        | 84       | 30 | 0.43                     | 225 | 81.06                    | 225 | 39.80                   | 236 | 11.95                      | 235 |
| stetson-p2        | 56       | 24 | 0.52                     | 148 | 38.41                    | 148 | 28.47                   | 154 | 11.10                      | 168 |
| stetson-p3        | 16       | 6  | 0.25                     | 8   | 3.74                     | 8   | 3.03                    | 8   | 1.53                       | 8   |
| vanbek-ad-opt-e   | 6        | 6  | 0.25                     | 13  | 5.47                     | 13  | 4.61                    | 13  | 2.17                       | 13  |
| xscsi-fifo2scsi   | 22       | 11 | 0.32                     | 45  | 12.02                    | 45  | 9.38                    | 46  | 4.12                       | 45  |
| xscsi-dma2fifo    | 18       | 9  | 0.28                     | 37  | 9.53                     | 37  | 7.62                    | 37  | 3.50                       | 37  |
| xscsi-fifo2dma    | 14       | 8  | 0.29                     | 20  | 7.87                     | 20  | 6.01                    | 20  | 2.78                       | 20  |
| xscsi-fifocellctl | 6        | 5  | 0.25                     | 10  | 5.48                     | 10  | 4.51                    | 10  | 2.10                       | 10  |
| yun-diffeq-alu2   | 32       | 14 | 0.32                     | 89  | 20.02                    | 89  | 16.21                   | 90  | 6.19                       | 89  |
| yun-diffeq-alu1   | 18       | 10 | 0.29                     | 38  | 13.28                    | 38  | 10.61                   | 38  | 4.67                       | 38  |
| yun-diffeq-mul1   | 8        | 7  | 0.26                     | 32  | 7.43                     | 32  | 6.04                    | 32  | 2.73                       | 32  |
| yun-diffeq-mul2   | 6        | 6  | 0.25                     | 13  | 5.74                     | 13  | 4.46                    | 13  | 2.06                       | 13  |

[29] implicit minimizer is exact in number of cubes, but does not perform literal minimization. The ESPRESSO-HF [29] minimizer is heuristic. It should be noted that the HFMIN, IMPYMIN, and ESPRESSO-HF tools are optimized for the more difficult problem of multi-output minimization. As our goal is to generate controllers mainly with small delay rather than small area, the minimizers are run in single-output configurations for the benchmarks presented in this article.

The runtime and literal comparisons shown in Tables VI

TABLE VII

Benchmark comparisons of standard gate implementations between existing two-level logic minimizers.

| Design            | $ \Phi $ | IO | ATACS<br>(literal-exact) |     | HFMIN<br>(literal-exact) |     | IMPYMIN<br>(cube-exact) |     | ESPRESSO-HF<br>(heuristic) |     |
|-------------------|----------|----|--------------------------|-----|--------------------------|-----|-------------------------|-----|----------------------------|-----|
|                   |          |    | Time                     | Lit | Time                     | Lit | Time                    | Lit | Time                       | Lit |
| ack-cdplayer-p1   | 88       | 48 | 80.36                    | 708 | impossible               |     | 1254.57                 | 743 | 85.46                      | 736 |
| ack-fibonacci     | 60       | 38 | 2.98                     | 415 | 3132.84                  | 415 | 346.50                  | 522 | 19.27                      | 494 |
| ack-diffeq        | 36       | 35 | 1.16                     | 261 | 142.56                   | 261 | 57.84                   | 287 | 10.81                      | 288 |
| ack-barcode       | 38       | 31 | 1.35                     | 229 | 105.50                   | 229 | 65.27                   | 254 | 9.02                       | 253 |
| ack-gcd           | 33       | 21 | 0.52                     | 98  | 18.74                    | 98  | 23.37                   | 98  | 4.04                       | 98  |
| ack-factorial     | 23       | 17 | 0.39                     | 40  | 11.57                    | 40  | 10.23                   | 40  | 4.59                       | 40  |
| cache-ctrl        | 98       | 36 | 238.48                   | 500 | 1316.79                  | 500 | 236.59                  | 550 | 18.58                      | 536 |
| chu-ad-opt-e      | 8        | 6  | 0.29                     | 11  | 3.62                     | 11  | 3.86                    | 11  | 2.19                       | 11  |
| dme-e             | 20       | 8  | 0.30                     | 22  | 6.30                     | 22  | 5.58                    | 22  | 2.13                       | 22  |
| dme-fast-e        | 20       | 8  | 0.31                     | 30  | 6.06                     | 30  | 5.68                    | 30  | 2.18                       | 30  |
| dram-ctrl         | 26       | 14 | 0.47                     | 50  | 8.93                     | 50  | 8.28                    | 50  | 3.10                       | 50  |
| hp-ir-sc-ctrl     | 76       | 30 | 3.11                     | 264 | 77.47                    | 264 | 41.57                   | 280 | 9.99                       | 298 |
| hp-ir-sd-ctrl     | 54       | 24 | 0.95                     | 131 | 29.91                    | 131 | 27.62                   | 134 | 7.64                       | 132 |
| hp-ir-it-ctrl     | 24       | 13 | 0.36                     | 52  | 10.48                    | 52  | 9.03                    | 53  | 3.00                       | 54  |
| hp-ir-rf-ctrl     | 26       | 13 | 0.38                     | 32  | 8.69                     | 32  | 7.37                    | 32  | 3.34                       | 32  |
| hp-ir-two-tick    | 18       | 8  | 0.30                     | 15  | 4.83                     | 15  | 4.19                    | 15  | 2.63                       | 15  |
| hp-ir             | 16       | 5  | 0.28                     | 8   | 3.21                     | 8   | 3.20                    | 8   | 1.56                       | 8   |
| postoffice-pesnd  | 28       | 10 | 0.40                     | 65  | 5.87                     | 65  | 6.12                    | 65  | 2.22                       | 65  |
| postoffice-bufsnd | 17       | 8  | 0.34                     | 34  | 5.54                     | 34  | 5.64                    | 34  | 2.15                       | 34  |
| postoffice-bufrd  | 14       | 7  | 0.29                     | 15  | 4.78                     | 15  | 4.92                    | 15  | 1.35                       | 15  |
| pscsi-pscsi       | 119      | 22 | 9.63                     | 400 | 53.42                    | 400 | 42.40                   | 421 | 8.33                       | 428 |
| pscsi-isend       | 22       | 11 | 0.39                     | 74  | 8.19                     | 74  | 8.08                    | 74  | 3.28                       | 75  |
| pscsi-irvc        | 13       | 9  | 0.35                     | 31  | 6.70                     | 31  | 5.60                    | 31  | 2.49                       | 31  |
| pscsi-trcv-bm     | 18       | 10 | 0.36                     | 40  | 8.18                     | 40  | 7.21                    | 40  | 3.87                       | 40  |
| pscsi-trcv        | 14       | 8  | 0.34                     | 25  | 4.73                     | 25  | 5.12                    | 25  | 2.71                       | 25  |
| pscsi-tsend-bm    | 24       | 11 | 0.39                     | 62  | 8.73                     | 62  | 7.60                    | 62  | 3.41                       | 62  |
| pscsi-tsend       | 24       | 11 | 0.39                     | 55  | 8.51                     | 55  | 7.74                    | 55  | 3.41                       | 56  |
| sscsi-isend-bm    | 24       | 11 | 0.40                     | 61  | 7.92                     | 61  | 7.22                    | 61  | 3.88                       | 61  |
| sscsi-isend-csm   | 18       | 11 | 0.39                     | 44  | 6.88                     | 44  | 6.09                    | 44  | 3.80                       | 44  |
| sscsi-trcv-bm     | 24       | 11 | 0.39                     | 54  | 7.34                     | 54  | 6.54                    | 54  | 3.85                       | 54  |
| sscsi-trcv-csm    | 18       | 11 | 0.38                     | 43  | 7.89                     | 43  | 6.05                    | 43  | 2.74                       | 43  |
| sscsi-tsend-bm    | 26       | 11 | 0.40                     | 64  | 7.50                     | 64  | 8.14                    | 64  | 3.80                       | 64  |
| sscsi-tsend-csm   | 22       | 11 | 0.39                     | 37  | 7.32                     | 37  | 7.01                    | 38  | 2.88                       | 39  |
| stetson-p1        | 84       | 30 | 4.06                     | 284 | 78.32                    | 284 | 42.95                   | 298 | 9.56                       | 315 |
| stetson-p2        | 56       | 24 | 0.96                     | 148 | 33.58                    | 148 | 28.17                   | 153 | 8.73                       | 155 |
| stetson-p3        | 18       | 7  | 0.30                     | 11  | 3.33                     | 11  | 3.78                    | 11  | 2.23                       | 11  |
| vanbek-ad-opt-e   | 6        | 6  | 0.29                     | 14  | 3.27                     | 14  | 3.79                    | 14  | 1.85                       | 14  |
| xscsi-fifo2scsi   | 26       | 11 | 0.38                     | 83  | 7.17                     | 83  | 8.50                    | 84  | 2.97                       | 83  |
| xscsi-dma2fifo    | 22       | 9  | 0.33                     | 52  | 6.35                     | 52  | 7.50                    | 52  | 2.96                       | 52  |
| xscsi-fifo2dma    | 16       | 8  | 0.35                     | 29  | 5.07                     | 29  | 4.61                    | 29  | 2.65                       | 29  |
| xscsi-fifocellctl | 6        | 5  | 0.29                     | 11  | 3.47                     | 11  | 3.17                    | 11  | 1.87                       | 11  |
| yun-diffeq-alu2   | 36       | 15 | 0.51                     | 143 | 14.43                    | 143 | 15.77                   | 156 | 5.34                       | 146 |
| yun-diffeq-alu1   | 18       | 10 | 0.33                     | 43  | 8.65                     | 43  | 8.18                    | 43  | 3.38                       | 44  |
| yun-diffeq-mul1   | 8        | 7  | 0.29                     | 42  | 4.62                     | 42  | 4.61                    | 42  | 2.44                       | 42  |
| yun-diffeq-mul2   | 6        | 6  | 0.29                     | 15  | 3.17                     | 15  | 3.95                    | 15  | 1.94                       | 15  |

and VII contains the largest benchmarks that have been built in the burst-mode community to date. The *postoffice* [4], *cache-ctrl* [23], *diffeq* [32], *cd-player* [12], *pscsi* [31], *sscsi* [24], *xscsi* [31], *dram-ctrl* [24], and *barcode* [25] are all derived from real-life designs. The burst-mode controllers *ack-cdplayer*, *ack-fibonacci*, *ack-diffeq*, *ack-barcode*, *ack-gcd*, and *ack-factorial* are generated automatically from a procedural language description by the high-level synthesis framework ACK [8], [9], [15], [16], [17]. The other examples are classic burst-mode benchmarks from various publications. All benchmarks are run on a 333 MHz Ultrasparc-2 processor with 1 GB of physical memory and 800 MB of virtual memory. All benchmarks that finished ran com-

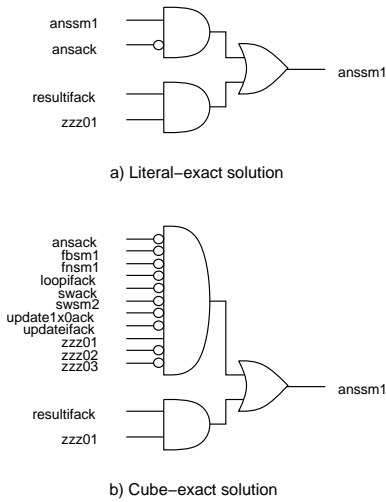


Fig. 16. Logic networks for *anssm1* in *ack-fibonacci* from a) ATACS literal-exact and b) IMPYMIN cube-exact algorithms.

pletely in physical memory. Hence the results should show the true runtime potential of the respective minimizers.

As can be seen in Table VI, for gC implementations, our logic minimization method, implemented in the tool ATACS, is very fast for all benchmarks. Our algorithm achieves sub-second synthesis even for the largest benchmarks and is over three orders of magnitude faster than the closest literal-exact solution for these benchmarks. As illustrated in Table VII, our logic minimization method is also very fast for the majority of the standard gate benchmarks. As illustrated by the *ack-cdplayer-p1*, *ack-fibonacci*, and *ack-diffeq* benchmarks our algorithm is especially efficient when minimizing large and complex extended burst-mode controllers because of the compact way our method represents concurrency and level signals. For these benchmarks, our method is over two orders of magnitude faster than the literal-exact tool HFMIN. The IMPYMIN tool performs substantially better runtime wise than HFMIN for the largest benchmarks but produces cube-exact solutions. The heuristic ESPRESSO-HF minimizer performs well for all benchmarks in terms of runtime. On average, IMPYMIN and ESPRESSO-HF both produce high quality solutions that are close to minimal for most benchmarks. However, since these tools are not based on literal-exact algorithms, their solutions can differ in number of literals by up to 10.5% and 13.5% respectively for gC, and 25% and 19% for standard gate implementations, compared to the literal exact solutions produced by HFMIN and ATACS.

Figure 16 illustrates the logic equations derived by ATACS and IMPYMIN for output *anssm1* of benchmark *ack-fibonacci*. While the cube-exact algorithm of IMPYMIN finds a minimum cube implementation, one of the cubes is not minimal in number of literals. The cube-exact solution requires a total of 13 literals to implement output *anssm1* compared to 4 literals for the literal exact solution.

These observations underscore the importance of achieving fast literal exact minimization. Even when comparing with the heuristic algorithm of ESPRESSO-HF, ATACS provides competitive runtimes and, in addition, produces lit-

eral exact covers. In addition to faster runtimes, our method can perform literal-exact minimization for designs where this has previously been impossible. For the *ack-cd-player-p1* benchmark, when targeting a gC implementation, HFMIN runs out of memory with 1.8 GB of total available memory. For the same benchmark our minimizer ATACS completes in 0.42 seconds using less than 15 MB of memory. It is worth noting that the one benchmark, *cache-ctrl*, for which ATACS has a relatively long runtime when targeting a standard gate implementation, has a large number of required cubes per output. The large number of required cubes for this benchmark results in a diminishing return for ATACS otherwise effective divide and merge minimization strategy, explaining the longer runtime. Nevertheless, for this benchmark, ATACS is still over five times faster than the only other literal exact algorithm HFMIN.

## IX. CONCLUSIONS

An efficient algorithm for single output literal-exact two-level logic minimization has been presented. The effectiveness of the presented logic minimizer can be mainly attributed to two factors. First, the signal concurrency properties of extended burst-mode controllers allows them to be expressed very efficiently in the form of compacted state graphs. Using compacted state graphs to represent extended burst-mode finite state machines, time spent in state graph exploration grows linearly, rather than exponentially, with the complexity (amount of signal concurrency and number of level signals) of the specification. The notion of compacted states is also exploited to significantly reduce the time spent in solving the binate and unate cover problems necessary to find hazard-free minimal solutions, as the size of the cover tables is significantly reduced. Second, the presented single-output minimization algorithm naturally divides the problem of finding a solution into smaller sub-problems of finding local unique solutions for each required cube separately, which are then merged into a final minimal solution for the entire output function. This divide and merge strategy has shown to perform well for most benchmarks and particularly where the number of required cubes is limited. Put together, these techniques form a literal-exact logic minimization approach that can perform several orders of magnitude faster than existing methods based on classical minimization approaches for large and complex controllers, making interactive and iterative design exploration possible.

This article addresses only the single-output minimization problem. If multiple outputs are minimized concurrently gate sharing can lead to reduced area requirements and may also improve performance by decreasing fanout counts on inputs. In the future, we would like to extend our method to the multi-output minimization problem.

## ACKNOWLEDGMENTS

The authors would like to thank Ganesh Gopalakrishnan and Prabhakar Kudva for helpful comments and discussions, and Michael Theobald and Steven Nowick for invaluable help with benchmarks and tools.

## REFERENCES

- [1] P. Beerel and T.-Y. Meng. Automatic gate-level synthesis of speed-independent circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 581–587. IEEE Computer Society Press, Nov. 1992.
- [2] P. A. Beerel, C. J. Myers, and T. H.-Y. Meng. Covering conditions and algorithms for the synthesis of speed-independent circuits. *IEEE Transactions on CAD*, Mar. 1998.
- [3] T.-A. Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*. PhD thesis, MIT Laboratory for Computer Science, June 1987.
- [4] B. Coates, A. Davis, and K. Stevens. The Post Office experience: Designing a large asynchronous chip. *Integration, the VLSI journal*, 15(3):341–366, Oct. 1993.
- [5] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*, E80-D(3):315–325, Mar. 1997.
- [6] R. M. Fuhrer. *Sequential Optimization of Asynchronous and Synchronous Finite-State Machines*. PhD thesis, Department of Computer Science, Columbia University, 1999.
- [7] R. M. Fuhrer, S. M. Nowick, M. Theobald, N. K. Jha, B. Lin, and L. Plana. Minimalist: An environment for the synthesis, verification and testability of burst-mode asynchronous machines. Technical Report TR CUCS-020-99, Columbia University, NY, July 1999.
- [8] H. Jacobson, E. Brunvand, G. Gopalakrishnan, and P. Kudva. High-level asynchronous system design using the ACK framework. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 93–103. IEEE Computer Society Press, Apr. 2000.
- [9] H. Jacobson and G. Gopalakrishnan. Application-specific programmable control for high-performance asynchronous circuits. *Proceedings of the IEEE*, 87(2):319–331, Feb. 1999.
- [10] K. W. James and K. Y. Yun. Average-case optimized transistor-level technology mapping of extended burst-mode circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 70–79, 1998.
- [11] S. T. Jung and C. J. Myers. Direct synthesis of timed asynchronous circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 332–337, Nov. 1999.
- [12] J. Kessels, K. van Berkel, R. Burgess, M. Roncken, and F. Schalij. An error decoder for the compact disc player as an example of VLSI programming. Technical report, Philips Research Laboratories, Eindhoven, The Netherlands, 1992.
- [13] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky. Analysis and identification of speed-independent circuits on an event model. *Formal Methods in System Design*, 4(1):33–75, 1994.
- [14] A. Kondratyev, M. Kishinevsky, and A. Yakovlev. Hazard-free implementation of speed-independent circuits. *IEEE Transactions on CAD*, 17(9):749–771, Sept. 1998.
- [15] P. Kudva. *Synthesis of Asynchronous Systems Targeting Finite State Machines*. PhD thesis, Computer Science Department, University of Utah, 1995.
- [16] P. Kudva, G. Gopalakrishnan, and H. Jacobson. A technique for synthesizing distributed burst-mode circuits. In *Proc. ACM/IEEE Design Automation Conference*, 1996.
- [17] P. Kudva, G. Gopalakrishnan, H. Jacobson, and S. M. Nowick. Synthesis of hazard-free customized CMOS complex-gate networks under multiple-input changes. In *Proc. ACM/IEEE Design Automation Conference*, 1996.
- [18] K.-J. Lin, C.-W. Kuo, and C.-S. Lin. Synthesis of hazard-free asynchronous circuits based on characteristic graph. *IEEE Transactions on Computers*, 46(11):1246–1263, Nov. 1997.
- [19] A. Marshall, B. Coates, and P. Siegel. Designing an asynchronous communications chip. *IEEE Design & Test of Computers*, 11(2):8–21, 1994.
- [20] D. E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *Proceedings of an International Symposium on the Theory of Switching*, pages 204–243. Harvard University Press, Apr. 1959.
- [21] C. J. Myers. *Computer-Aided Synthesis and Verification of Gate-Level Timed Circuits*. PhD thesis, Dept. of Elec. Eng., Stanford University, Oct. 1995.
- [22] S. M. Nowick. *Automatic Synthesis of Burst-Mode Asynchronous Controllers*. PhD thesis, Stanford University, Department of Computer Science, 1993.
- [23] S. M. Nowick, M. E. Dean, D. L. Dill, and M. Horowitz. The design of a high-performance cache controller: a case study in asynchronous synthesis. *Integration, the VLSI journal*, 15(3):241–262, Oct. 1993.
- [24] S. M. Nowick, K. Y. Yun, and D. L. Dill. Practical asynchronous controller design. In *Proc. International Conf. Computer Design (ICCD)*, pages 341–345. IEEE Computer Society Press, Oct. 1992.
- [25] P. R. Panda and N. Dutt. 1995 high level synthesis design repository. Technical Report 95-04, University of California, Irvine, U.S.A., 1995.
- [26] E. Pastor, J. Cortadella, A. Kondratyev, and O. Roig. Structural methods for the synthesis of speed-independent circuits. *IEEE Transactions on CAD*, 17(11):1108–1129, Nov. 1998.
- [27] J. Rutten and M. Berkelaar. Efficient exact and heuristic minimization of hazard-free logic. In *Proc. International Conf. Computer Design (ICCD)*, pages 152–159, Oct. 1998.
- [28] K. S. Stevens, S. Rótem, R. Ginosar, P. Beerel, C. J. Myers, K. Y. Yun, R. Koi, C. Dike, and M. Roncken. An asynchronous instruction length decoder. *IEEE Journal of Solid-State Circuits*, 36(2):217–228, Feb. 2001.
- [29] M. Theobald and S. M. Nowick. Fast heuristic and exact algorithms for two-level hazard-free logic minimization. *IEEE Transactions on CAD*, 17(11):1130–1147, Nov. 1998.
- [30] C. Ykman-Couvreur, B. Lin, and H. de Man. Assassin: A synthesis system for asynchronous control circuits. Technical report, IMEC, Sept. 1994. User and Tutorial manual.
- [31] K. Y. Yun. *Synthesis of Asynchronous Controllers for Heterogeneous Systems*. PhD thesis, Stanford University, Aug. 1994.
- [32] K. Y. Yun, P. A. Beerel, V. Vakilotojar, A. E. Dooply, and J. Arceo. The design and verification of a high-performance low-control-overhead asynchronous differential equation solver. *IEEE Transactions on VLSI Systems*, 6(4):643–655, Dec. 1998.
- [33] K. Y. Yun and D. L. Dill. Automatic synthesis of extended burst-mode circuits: Part I (specification and hazard-free implementation). *IEEE Transactions on CAD*, 18(2):101–117, Feb. 1999.

**Hans M. Jacobson** received his M.S. degree in Computer Engineering from Lulea University, Sweden, in 1996. He is currently a graduate student at University of Utah, Salt Lake City, UT, and is performing his dissertation research on Interlocked Synchronous Pipelines at IBM T.J. Watson Research Center, Yorktown, NY. His interests include novel clocking techniques for high speed, low power microprocessors and ASICs, simultaneous multi-threaded microprocessor architectures, and asynchronous circuit and system design. He is a recipient of the 1999 University of Utah Graduate Research Fellowship Award.

**Chris J. Myers** received the B.S. degree in electrical engineering and Chinese history in 1991 from the California Institute of Technology, Pasadena, CA, and the M.S.E.E. and Ph.D. degrees from Stanford University, Stanford, CA, in 1993 and 1995, respectively. He is an Associate Professor in the Department of Electrical and Computer Engineering, University of Utah, Salt Lake City, UT. His current research interests are innovative architectures for high performance and low power, algorithms for the computer-aided analysis and design of real-time concurrent systems, analog error control decoders, formal verification, and asynchronous circuit design. Dr. Myers received an NSF Fellowship in 1991, an NSF CAREER award in 1996, and a best paper award at Async99.