An Abstraction Algorithm for the Verification of Level-Sensitive Latch-Based Netlists

Jason Baumgartner[1], Tamir Heyman[2], Vigyan Singhal[3], and Adnan Aziz[4]

[1] IBM Enterprise Systems Group, Austin, Texas 78758, USA
jasonb@austin.ibm.com
[2] IBM Haifa Research Laboratory, Haifa, Israel
tamirh@cs.technion.ac.il
[3] Tempus Fugit, Inc., Fremont, California 94538, USA
vigyan@tempusf.com
[4] The University of Texas, Austin, Texas 78712, USA
adnan@ece.utexas.edu

Abstract. High-performance hardware designs often intersperse combinational logic freely between level-sensitive latch layers (wherein each layer is transparent during only one clock phase), rather than utilizing master-slave latch pairs with no combinational logic between. While such designs may generally achieve much faster clock speeds, this design style poses a challenge to verification. In particular, unless the k-phase netlist N is abstracted to a full-cycle register-based netlist N', verification of N requires k times (or greater) as many state variables as would be necessary to obtain equivalent verification of N'. We present algorithms to automatically identify and abstract k-phase netlists – i.e., to perform phase abstraction – by selectively eliminating latches. The abstraction is valid for model checking CTL* formulae which reason solely about latches of a single phase. This algorithm has been implemented in the model checker *RuleBase*, and used to enhance the model checking of IBM's Gigahertz Processor, which would not have been feasible otherwise due to computational constraints. This abstraction has furthermore allowed verification engineers to write properties and environments more efficiently.

This paper extends results reported by the authors at the following conference:

– J. Baumgartner, T. Heyman, V. Singhal, and A. Aziz. Model Checking the IBM Gigahertz Processor: An Abstraction Algorithm for High-Performance Netlists. In *Proceedings of the Conference on Computer-Aided Verification*, July 1999.

1 Introduction

A $latch$ is a hardware memory element with two Boolean inputs – data and clock – and one Boolean output. High performance netlists often use level-sensitive latches [1], as balancing combinational logic between such latches may offer performance superior to that of a netlist comprised of edge-sensitive registers. For such a latch, when its clock input is a certain value (e.g., a logical "1"), the value at its data input will be combinationally propagated to its data output (a condition termed *transparent mode*); otherwise, its last propagated value is held.

In common two-phase designs, the clock is modeled as a signal which alternates between 0 and 1 at each time-step. A latch which is transparent when the clock is a 0 will be denoted as a ϕ_0 latch (often referred to as an *L1* latch); one which is transparent when the clock is a 1 will be denoted as ϕ_1 (often referred to as an *L2* latch). Hardware design rules, arising from timing constraints, require any structural path between two ϕ_0 latches to pass through a ϕ_1 latch, and vice-versa. An elementary design style requires each ϕ_0 latch to fan out directly to a ϕ_1 latch (called a master-slave latch pair), and allows only ϕ_1 to drive combinational logic. However, a common high-performance hardware design technique involves utilizing combinational logic freely between ϕ_0 and ϕ_1 latches to better utilize each half-cycle of the clock. Such designs are typically explicitly implemented in this manner; this structure is not the byproduct of a synthesis tool, but instead a necessary technique to ensure the highest performance design.

There are two major difficulties with the verification of k-phase netlists. The first is intrinsic; because of the larger number of state elements (one per latch) as compared to a functionally equivalent full-cycle (i.e., edge-sensitive register-based) netlist, the verification toolset often requires much more time and memory. Additionally, since k image computations are necessary per clock period, the diameter of such a netlist is k times that of an equivalent full-cycle netlist. The second is methodological; the manual modeling of environments and properties is often more complicated in that they must be written in terms of the less abstract k-phase netlist, and an oscillating clock must be explicitly introduced which is in the support of all state variables. Therefore, we propose the technique of *phase abstraction* to overcome these difficulties.

There are two methodological approaches possible using the techniques to be discussed in this paper. The first, termed Methodology 1, is to make the user explicitly aware of the process of phase abstraction – the abstraction is performed only on the netlist under test, the user explicitly specifies properties and environments in terms of the phase-abstracted netlist, and any resulting traces will be in terms of the phase-abstracted netlist. The second, termed Methodology 2, is to perform phase abstraction "invisibly" under the covers. The user writes their environment as a k-phase netlist, and the specification is synthesized (automatically or manually) into a k-phase netlist implementing a corresponding automaton; both of these components are then composed onto the netlist under test and phase abstraction is performed on this composition. This phase abstracted composition is then passed into a verification tool. Any resulting trace from the verification tool may then automatically be "un-phase-abstracted" before being presented to a user. Both methods benefit from the reduction in state variables and diameter. Only Methodology 1 benefits from the ability to model more abstract specification – in many cases this implies that Methodology 1 is preferred. However, if a user

must reason about latches of multiple phases in their specification, Methodology 2 is often preferred to ensure soundness and completeness.

In this paper we develop a set of efficient structural algorithms for *phase abstracting* a k-phase netlist N to a full-cycle netlist N', which may be utilized for enhanced verification in any state-exploration-based framework (e.g., model checking, semi-formal algorithms, simulation). We define a *k-phase bisimulation* equivalence between the abstracted and unabstracted netlists – refer to Theorems 2-4. This equivalence ensures that specification written in terms of ϕ_{k-1} latch outputs need not be modified other than a conversion to full-cycle format (as will be discussed in Section 4); such conversion may be performed either manually using Methodology 1 or automatically using Methodology 2. Our technique provably performs maximal such reductions for two-phase netlists (refer to Theorem 5), and thus provides an important model reduction step which may greatly augment existing techniques (such as cone-of-influence, retiming [2], etc.). As we demonstrate, this abstraction alone reduces the number of state variables (and the diameter) of the netlist by a factor of $1/k$, and possibly much greater. Additionally, designers and verification engineers often prefer to reason about full-cycle netlists, hence often prefer to write specification, and view counterexamples, in terms of the abstracted netlist. Methodology 1 of this abstraction has been implemented in the model checker *RuleBase* [3], and has greatly enhanced the model checking of IBM's Gigahertz Processor which, as our experimental evidence demonstrates, would not have been feasible otherwise. Furthermore, our linear-time algorithms have proven very useful to quickly identify latch and clock connectivity errors, as we discuss in Section 6.

A similar technique was proposed in [4] for sequential hardware equivalence. They propose globally converting latches of all but a single phase into buffers – single-input AND vertices. Their work proved correctness for the steady-state subgraph of the abstracted netlist; as such, initial values are discarded. However, this approach is insufficient for model checking; modern hardware designs typically require an explicit initialization sequence (e.g., via scan chains) before proper functionality is ensured. Failure to consider initial values (and transitions outside of the steady state) in model checking may fail to expose certain arbitrarily complicated design flaws occurring before steady state is reached, and even prevent the netlist from reaching its intended steady state. Our approach does preserve initial values, and we prove an equivalence between the original and abstracted netlists relative to their initial states. The technique in [4] further proposed a globally greedy approach of "removing all but the smallest phase set" of latches. They propose retiming as a second reduction step to ensure minimal latch counts. Our work calculates minimally-sized partitions of the netlist during its topological analysis, and allows a greedy choice of which phases to discard for each partition, independently of the other partitions, hence is able to achieve reductions beyond those possible with the technique of [4] alone (without the more costly retiming step).

Many hardware compilers allow automatic translation of master-slave latch sets into a single edge-sensitive register. Retiming algorithms [5] may be used to retime the netlist such that ϕ_0-ϕ_{k-1} layers become adjacent and one-to-one. However, retiming adds a degree of complexity to a verification toolset in that the specification and any witness and counterexample traces may need to be retimed as well to match the retimed netlist, which becomes rather impractical for Methodology 1. Model checking further

requires preservation of initial values, which may necessitate extra processing in the case of retiming [6]. A detailed exposition of retiming register-based netlists while verifying $\mathsf{AG}(p)$ properties is provided in [2] – their approach is akin to our Methodology 2. Overall, retiming requires quadratic resources or greater[1]; in contrast, our technique is a linear-time algorithm: $O(|vertices| + |edges|)$. We therefore conclude that the use of retiming to perform phase abstraction is inferior to our approach.

Note that phase abstraction and retiming offer somewhat orthogonal benefits; phase abstraction reduces latch count along any directed cycle by a factor of $1/k$, which retiming cannot alter. Conversely, retiming attempts to globally relocate state elements to minimize their total count; our technique performs some degree of such minimization in that it greedily eliminates latches per partition, but overall has a smaller solution space to choose from than retiming as may be inferred by its linear complexity. It is our experience that performing phase abstraction as a pre-process prior to importing the resulting register-based netlist into a verification toolset allows for a simpler tool implementation (since register clocks may be treated as implicit), and yields computationally superior results even if retiming is subsequently applied to the smaller phase-abstracted netlist as noted in [4].

A similar concept is the notion of *stuttering bisimulation* [9], which relates two machines which are semantically equivalent except that either may add "repetitious state transitions" that do not appear in the other. The satisfaction of two stuttering bisimilar states is identical for CTL* formulae with no $\mathsf{X}(f)$ subformulae [10]. Stuttering bisimilarity offers some degree of insight into the nature of our k-phase bisimilarity. Indeed, k-phase abstraction yields an abstracted machine M related to its original M' by a k-stuttering – each state of M has k stuttering-bisimilar correspondents in M'. Beyond its relation to stuttering bisimilarity, our work provides linear-time structural netlist analysis and abstraction algorithms to identify and abstract k-phase netlists, demonstrates applicability of CTL* formulae including $\mathsf{X}^k(f)$, and preserves bisimilarity under composition. This relation will be discussed further in Section 4.1.

Related to the topic of phase abstraction is c-slow abstraction [11]. Like a c-phase netlist, the topology of c-slow netlists guarantee that any directed cycle has modulo-c state elements; similar c-coloring may be applied to both netlist types. However, unlike phase abstraction, c-slow abstraction is only applicable to netlists composed of registers. Furthermore, c-slow netlists generally do not stutter whatsoever, and all of their initial values have semantic importance, unlike c-phase netlists. As with retiming, use of c-slow abstraction after phase abstraction may be a beneficial verification strategy.

The work of [12] provides a methodology for a specification to operate at a different time-scale as a hardware implementation to increase the utility of assume-guarantee reasoning; our Methodology 1 could be viewed as such a paradigm. However, they do not focus on abstractions to enhance verification, only on the mechanics of interfacing a specification in one time-scale to an implementation in another. The work of [13] provides a general set of formalisms to relate various transformations of netlists with various latching and clocking schemes, such as multi-phase netlists, retimed netlists,

[1] Retiming is solvable as a min-cost flow problem [7], for which one of the most efficient known algorithms is the *enhanced capacity scaling algorithm*, which is $O\big(|edges| \cdot log(|vertices|) \cdot (|edges| + |vertices| \cdot log(|vertices|))\big)$ [8].

and netlists with multiple clock domains. In focusing more on a general framework, they do not address implications of the transformations in a property checking environment or on reduction optimality.

This paper is organized as follows. We provide a formal definition of the syntax and semantics of netlists in Section 2. In Section 3 we introduce a generic two-phase netlist, and two abstracted full-cycle variants of this netlist. In Section 4 we prove that our k-phase abstraction yields a k-phase bisimilar netlist, that this bisimilarity preserves CTL* model checking including X^k, and that this bisimilarity is preserved under composition. In Section 5 we introduce the algorithms used to perform phase abstraction, and discuss their optimality. In Section 6 we provide experimental results of our algorithms as implemented in RuleBase [3] for application to IBM's Gigahertz Processor. Several implementation details, including trace generation and CTL* expressibility, are discussed in Section 7. We conclude this paper in Section 8.

2 Netlists: Syntax and Semantics

In this section we define the syntax and semantics of netlists. Those well-versed in such topics may wish to proceed to Definition 12.

Definition 1. A *netlist* is a tuple $\tilde{N} = \langle\langle V, E\rangle, T, Z\rangle$. The tuple $\langle V, E\rangle$ is a finite directed graph, where the vertices V represent *gates*, and the edges E represent *interconnections*.[2] Function $T : V \mapsto types$ defines the semantic gate *type* associated with each gate v. Function $Z : V \mapsto V$ is the initial value mapping $Z(v)$ of each gate v.

Our gate *types* define a set of constants, combinational gates, primary inputs, and latches, whose semantics will be provided in Definition 2. Note that the type of a gate will generally place constraints upon its indegree – for example, constants and primary inputs have indegree of zero, and inverters have indegree of one. Latches have indegree of two: their inputs are *data* and *clock*. We refer to the source vertex of the clock input edge of a latch v as *clock*(v), and of the data input as *data*(v). The initial value mappings have semantic importance only for latch types; this function is ignored for other vertices. We assume that the netlist contains no combinational cycles, and that the domain of Z is strictly combinational – i.e., the fanin cone of each $Z(v)$ contains no latches. We will refer hereafter to the set of latches $L = \{u \in V : T(u) = latch\}$, and the set of inputs $I = \{u \in V : T(u) = input\}$.

Definition 2. The *semantics of a netlist* $\tilde{N}$ are defined in terms of semantic traces: $0, 1$ valuations to gates over time. We denote the set of all legal traces associated with a netlist by $P \subseteq [V \times \mathbb{N} \mapsto \{0, 1\}]$, defining P as the subset of all possible functions from $V \times \mathbb{N}$ to $\{0, 1\}$ which are consistent with the following rule. The value of gate u

[2] Throughout the text we refer to *interconnections* as *nets*. This is somewhat imprecise; a net may have multiple sinks and sources, and is not necessarily "directed," whereas an edge has exactly one source and one sink, and is directed. We may transform arbitrary netlists by introducing appropriate single-output *net source* vertices, and single-input *net sink* vertices, to split nets to yield a semantically-equivalent graph-based representation. Our use of the term *net* unambiguously refers to a corresponding single-source, single-sink edge.

at time i in trace p is denoted by u_p^i. We define the value of a net $n = (u, v)$ at time i by its source vertex: $n_p^i = u_p^i$.

$$u_p^i = \begin{cases} s_{u_p}^i & : \ u \text{ is an input with sampled value } s_{u_p}^i \text{ at time } i \\ T_u(v_{1_p}^i, \ldots, v_{n_p}^i) & : \ u \text{ is a combinational gate with function } T_u = T(u) \\ \big(data(u)\big)_p^i & : \ u \text{ is a latch and } \big(\mathit{clock}(u)\big)_p^i = 1 \\ \big(Z(u)\big)_p^0 & : \ u \text{ is a latch and } i = 0 \text{ and } \big(\mathit{clock}(u)\big)_p^0 = 0 \\ u_p^{i-1} & : \ u \text{ is a latch and } i > 0 \text{ and } \big(\mathit{clock}(u)\big)_p^i = 0 \end{cases}$$

Term v_j denotes the source vertex of the j-th incoming edge to u, implying $(v_j, u) \in E$.

We now introduce some terminology related to describing the structure of netlists.

Definition 3. We define $\mathit{inlist}(U) = \{v : \exists u \in U.\ (v, u) \in \mathcal{E}\}$ as the set of vertices sourcing input edges to vertex set U. We define the indegree of a vertex set U by $\mathit{indegree}(U) = |\mathit{inlist}(U)|$.

Definition 4. We define *fanin cone*$(U) = U \cup$ *fanin cone*$\big(\mathit{inlist}(U)\big)$ for vertex set U.

We may analogously define *outlist* and *fanout cone*. We define the clock logic of a netlist as $D =$ *fanin cone*$\big(\mathit{clock}(L)\big)$, and assume without practical loss of generality that $\langle D, V \setminus D\rangle$ forms a cut of the netlist whose crossing edges are *clock* edges.

Definition 5. The *cone of influence* of a vertex set U is denoted as *coi*(U), and defined as *fanin cone*$(U) \cup$ *fanin cone*$\big(Z(L \cap$ *fanin cone*$(U))\big)$.

Note that the cone-of-influence of a vertex v is a superset of *fanin cone*(v), adding the fanin cones of initial values of latches which appear in the fanin cone of v.

Definition 6. The *combinational fanin* of vertex set U is defined as $\bigcup_{u \in U}$ *cfi*(u), where *cfi*(u) is defined as u if $u \in L$, else $u \cup$ *combinational fanin*$\big(\mathit{inlist}(u)\big)$ if $u \notin L$.

Definition 7. The *combinational fanout* of vertex set U is defined as $\bigcup_{u \in U}$ *cfo*(u), where *cfo*(u) is defined as *outlist*$(u) \cup$ *combinational fanout*$\big(\mathit{outlist}(u) \setminus L\big)$.

Intuitively, the combinational fanin cone of v contains all vertices in the fanin cone of v which may be reached without passing through a latch, and the combinational fanout cone of v contains all vertices in the fanout cone of the sinks of v which may be reached without passing through a latch.

Definition 8. A *k-phase netlist* N is a tuple $\langle \tilde{N}, \Phi, C\rangle$ where $\tilde{N}$ is a netlist, Φ represents a "global" clock, and $C : V \setminus D \mapsto 0, \ldots, k-1$ is a *k-coloring* function. Semantically, Φ acts an unconditional $\bmod k$ up counter which initializes to 0, thus $\Phi_p^i = i \mod k$ for any $p \in P$, extending the notion of a trace to k-phase netlists in the obvious manner, indicating which phase of latches are transparent at time-step i.

We require that $\big(\mathit{clock}(v)\big)_p^i = \big(\Phi_p^i = C(v)\big)$ for each $v \in L$ and each $p \in P$. For each vertex v, if $C(v) = j$, then $C\big(\{u : u \in$ *combinational fanout*$(v) \wedge u \in L\}\big) = (j+1) \mod k$ and also $C\big(\{u : u \in$ *combinational fanin*$(v) \wedge u \in L\}\big) = j$. We require that $C(I) = k - 1$.

Use of an integer rather than a set of Boolean values for Φ is merely a notational shorthand. Hereafter, we assume that a netlist is k-phase; before phase abstraction $k \geq 2$, and after phase abstraction we obtain a *full-cycle* netlist with $k = 1$ composed of registers. A linear-time algorithm may be used to color the nets of N; Φ further provides a "seed" for the coloring. The coloring of a k-phase netlist ensures that Definition 2 is well-formed; otherwise, a netlist may be prone to combinational cycles if, for example, any directed cycle contains only latches of a single color.

We use the following definitions to associate a netlist to a Moore machine [14] and its associated Kripke structure (similar to Grumberg and Long [15]).

Definition 9. A *Moore machine associated with a netlist* is a tuple $\langle N, L, I, S, S_0, O, \gamma, \delta \rangle$, where

- N is a k-phase netlist,
- L is the set of *state variables*,
- I is the set of *inputs*,
- $S = 2^{L \cup \Phi}$ is the set of *states*,
- $S_0 \subseteq S$ is the set of *initial states*, defined as $\bigcup_{p \in P} \{\{u \in L : \big(Z(u)\big)_p^0 = 1\}\}$,
- $O = \{u : C(u) = k-1\} \setminus \{u : I \cap$ *combinational fanin*$(u) \neq \emptyset\}$ is the set of *visible vertices*,
- $\gamma : S \mapsto 2^O$ is the *labeling function* as consistent with Definition 2, and
- $\delta \subseteq S \times 2^I \times S$ is the *transition relation* as consistent with Definition 2.

We uniquely identify each state by a subset of the state variables; intuitively, this subset represents those vertices which evaluate to a 1 in that state. We similarly define the labeling function as those color-$k-1$ vertices, minus vertices with primary inputs in their combinational fanin, which evaluate to 1 in the corresponding state. This set correlates to the set of outputs; its limitation on inputs is intrinsic in the definition of a Moore machine (a Mealy machine merely removes this restriction), and is necessary in general for compositionality as well as soundness of our abstraction. The transition relation is defined as those tuples (s, j, t) such that $\exists p \in P. \exists i \in \mathbb{N}.\ (s = \{u : u \in L \wedge u_p^i = 1\} \wedge j = \{u : u \in I \wedge u_p^i = 1\} \wedge t = \{u : u \in L \wedge u_p^{i+1} = 1\})$. We use $\Phi(s)$ as shorthand to represent the integer value of Φ in state s.

Definition 10. The *composition of Moore machines* M and M', denoted by $M \parallel M'$, is another machine M''. Netlists N and N' of M and M', respectively, may share inputs upon composition, and outputs of one may be inputs to another; otherwise their vertices are distinct. This implies that $\{V \setminus I\} \cap \{V' \setminus I'\} = \emptyset$ and $\{V \setminus \{O \cup I\}\} \cap I' = \emptyset$ and $\{V' \setminus \{O' \cup I'\}\} \cap I = \emptyset$. Furthermore, we require that $\Phi = \Phi'$, and that *fanin cone*$\big(Z''(L'')\big)$ remain combinational. M'' is defined as follows.

- $N'' = N \parallel N'$,
- $L'' = L \cup L'$,
- $I'' = \{I \cup I'\} \setminus \{O \cup O'\}$,
- $S'' = S \times S'$,
- $S_0'' = \bigcup_{p \in P''} \{\{u \in L'' : \big(Z''(u)\big)_p^0 = 1\}\}$,
- $O'' = \{u \in V'' : C(u) = k-1\} \setminus \{u \in V'' : I'' \cap$ *combinational fanin*$(u) \neq \emptyset\}$,

- $\gamma'' = \gamma \cup \gamma'$, and
- $\delta''\big((s,s'),x'',(t,t')\big)$ iff $\delta(s,\{x'' \cup \gamma'(s')\} \cap I, t)$ and $\delta'(s',\{x'' \cup \gamma(s)\} \cap I', t')$.

Since visible vertices and inputs all have color $k-1$, the composition of k-phase netlists N and N' yields a k-phase netlist N'' which inherits coloring, hence $C'' = C \cup C'$. Note that composition may increase the set of visible vertices since some inputs may be eliminated, hence $I'' \subseteq I \cup I'$ and $O'' \supseteq O \cup O'$. Note also that some cross-products of initial states of $M \parallel M'$ may be illegal, thus $S_0'' \subseteq S_0 \times S_0'$. Composition of Moore machines cannot introduce combinational cycles because their outputs are not combinational functions of their inputs.

Definition 11. The *Kripke structure associated with a Moore machine* is a tuple $\langle M, S^K, S_0^K, \mathcal{A}, \mathcal{L}, R\rangle$, where M is a Moore machine, $S^K = 2^{L \cup I \cup \Phi}$ is the set of *states*, $S_0^K = \{s \in S^K : s \setminus I \in S_0\}$ is the set of *initial states*, $\mathcal{A} = O$ is the set of *atomic propositions*, $\mathcal{L} = S^K \mapsto 2^O$ is the *labeling function*, and $R\big((s,x),(t,y)\big) \Leftrightarrow \delta(s,x,t)$ is the *transition relation*.

Intuitively, we define S_0^K as the subset of S^K which, when projected down to the latches, is equivalent to S_0. As per this definition, we assume that properties may only refer to color-$k-1$ nets which have no primary inputs in their combinational fanin (see Section 7.3 for a discussion of variants); note that elements of the clock logic are thus not visible. The color-$k-1$ restriction is intrinsic to phase abstraction. The restriction on nets with inputs in their combinational fanin is necessary due to the fact that inputs only propagate (i.e., affect transitions of the machine) during one time-step per clock period, whereas the lack of latches in their fanin does not prevent them from toggling every time-step. In other words, the inputs need not stutter, whereas all latch outputs must stutter. Restricting inputs to toggle only every k-th time-step is almost always desirable in practice since the netlist will likely be composed with other k-phase partitions, or occur at chip boundaries where sampling occurs only once per clock period. The restriction on clock logic is due to the fact that phase abstraction will discard the clock logic and replace it with an implicit clock. In the sequel we will use M to denote the Moore machine as well as the structure for the machine. Similarly, we will use N to refer to a netlist in addition to its respective Moore machine and structure.

The optimality of our algorithm results from the identification of *minimal dependent layers (MDLs)* of latches, and preserving only one phase of latches per MDL.

Definition 12. A *dependent layer* is a nonempty set of latches which satisfy the following rule. Denoting the $\phi_0, \ldots, \phi_{k-1}$ latches which comprise the dependent layer as $l_0, \ldots, l_{k-1}$, respectively, we require that l_{j+1} is a superset of all latches in the combinational fanout of l_j, and that l_j is a superset of all latches in the combinational fanin of $data(l_{j+1})$ for $0 \leq j < k-1$.

Definition 13. A dependent layer l is termed *minimal* if and only if there does not exist a nonempty set of latches l' which may be removed from l and still result in a nonempty dependent layer $l \setminus l'$.

Lemma 1. There is a unique MDL partition of any netlist.

Proof. We prove this lemma by contradiction. Let Q_0 and Q_1 be two non-equivalent partitions of N into MDLs. Let q_{0_i} represent the i-th MDL in Q_0, and q_{1_i} the i-th MDL in Q_1. For Q_0 to be non-unique, there must exist a q_{1_i} which is not an element of Q_0. Note that there cannot exist a q_{0_i} which is a superset of this q_{1_i} else q_{0_i} is not minimal (or is equivalent to q_{1_i}); similarly, this q_{1_i} cannot be a superset of any q_{0_i}.

If q_{1_i} is a singleton $\{a\}$ of color j, then there must exist no other latches in the fanout (unless $j = k-1$) or fanin (unless $j = 0$) cone of a else q_{1_i} is not a dependent layer. Clearly, the q_{0_i} which contains a is not minimal since the nonempty set $q_{0_i} \setminus \{a\}$ is a dependent layer – as is the singleton $\{a\}$.

If q_{1_i} has cardinality greater than one, there must exist two latches a and b in q_{1_i} such that $a \in q_{0_i}$ and $b \in q_{0_j}$ for $i \neq j$. Let $l_c, \forall c \in [0,k)$ be latch sets which are initially empty. If a is a ϕ_c, we add a to l_c. We next iteratively add to the latch sets as follows: for each $0 \leq d < k-1$, we add all ϕ_{d+1} latches in the combinational fanout of l_d to l_{d+1}. Similarly, for each $0 \leq d < k-1$, we add all ϕ_d latches in the combinational fanin of $data(l_{d+1})$ to l_d. If we perform an iteration of this process for each $d \in [0,k)$ and encounter no new latches, we have reached a fixed point of dependent latches. This process represents our MDL partitioning algorithm depicted in Figure 5. Note that each l_c must be a subset of q_{0_i}, else q_{0_i} is not a dependent layer. Furthermore, b must be one of the latches reached in this fixed point, else q_{0_i} is not minimal. This contradicts the claim that $q_{0_i} \neq q_{1_i}$.

Consider the two-phase netlist in Figure 1 (the triangles denote combinational logic, and the rectangles denote latches). The ϕ_0 latches are shaded. The two unique MDLs are marked with dotted boxes. Merely removing all ϕ_0 or all ϕ_1 latches will not yield an optimum reduction for this netlist; the ϕ_0 latches of layer A, and the ϕ_1 latches of layer B should be removed to yield an optimum solution of two latches.

We have hitherto assumed that our netlist is a Moore machine, which may be limiting in practice. One purpose for the restriction is compositionality; the composition of Mealy machines requires care to prevent the introduction of combinational cycles. However, we may readily view any Mealy k-phase netlist as the composition of k-phase Moore machines, with an additional combinational component with Moore machines and inputs in its combinational fanin. However, we do not allow vertices with primary inputs in their combinational fanin to be visible to specification, as this may generally become unsound when performing CTL* model checking with phase abstraction.

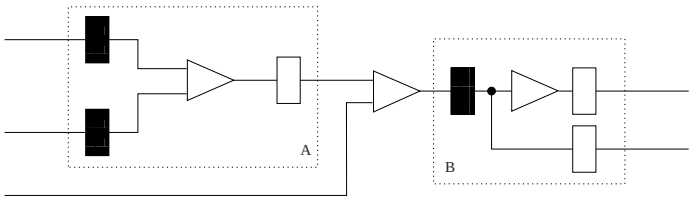

Fig. 1: Example Netlist with Two Minimal Dependent Layers

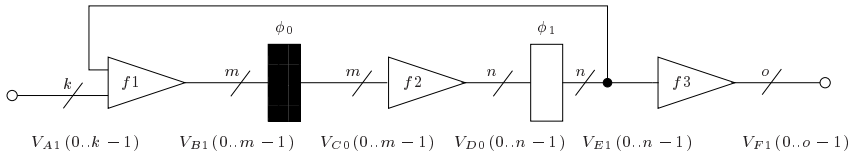

Fig. 2: Two-Phase Netlist N

Net	Time-step 0	Time-step 1
Φ	0	1
V_{A1}	a	a'
V_{B1}	$f1(a, Z(V_{E1}))$	$f1(a', Z(V_{E1}))$
V_{C0}	$f1(a, Z(V_{E1}))$	$f1(a, Z(V_{E1}))$
V_{D0}	$f2(f1(a, Z(V_{E1})))$	$f2(f1(a, Z(V_{E1})))$
V_{E1}	$Z(V_{E1})$	$f2(f1(a, Z(V_{E1})))$
V_{F1}	$f3(Z(V_{E1}))$	$f3(f2(f1(a, Z(V_{E1}))))$

Table 1: Symbolic Reachability Analysis of N – Base Case

Therefore, the only practical limitation of our technique to Mealy machines is this latter point – thus precluding reasoning about this combinational Mealy component.

3 Level-Sensitive Latch-Based versus Phase-Abstracted Netlists

Consider the two-phase netlist shown in Figure 2. Though we have defined netlists at the bit level, our examples are depicted in vectored form as a notational shorthand. This netlist has k primary inputs V_{A1}. The number associated with these labels refers to the color of the net (equivalent to that of its source vertex). The inputs fan out to $f1$, which is a function of V_{A1} and V_{E1}. The output of $f1$ is V_{B1} (an m-bit vector), which fans out to ϕ_0 latches that drive V_{C0}. Vector V_{C0} fans out to function $f2$, the output of which is n-bit vector V_{D0}. Vector V_{D0} fans out to ϕ_1 latches, which drive V_{E1}. The primary output V_{F1} is an o-bit combinational function $f3$ of V_{F1}. The symbolic simulation of this netlist is illustrated in Tables 1 and 2. Table 1 illustrates the initial two time-steps. Table 2 shows an "inductive" simulation beginning at any arbitrary "even" time-step j, hence the clock is a 0, meaning that the ϕ_0 latches are transparent.

The initial values of the latches are $Z(V_{C0})$ and $Z(V_{E1})$. Let Φ denote the global clock, which initializes to 0, and alternates between 0 and 1 at every time step, indicat-

Net	Time-step j	Time-step $j+1$	Time-step $j+2$	Time-step $j+3$
Φ	0	1	0	1
V_{A1}	b	b'	c	c'
V_{B1}	$f1(b, V_1)$	$f1(b', f2(f1(b, V_1)))$	$f1(c, f2(f1(b, V_1)))$	$f1(c', f2(f1(c, f2(f1(b, V_1)))))$
V_{C0}	$f1(b, V_1)$	$f1(b, V_1)$	$f1(c, f2(f1(b, V_1)))$	$f1(c, f2(f1(b, V_1)))$
V_{D0}	$f2(f1(b, V_1))$	$f2(f1(b, V_1))$	$f2(f1(c, f2(f1(b, V_1))))$	$f2(f1(c, f2(f1(b, V_1))))$
V_{E1}	V_1	$f2(f1(b, V_1))$	$f2(f1(b, V_1))$	$f2(f1(c, f2(f1(b, V_1))))$
V_{F1}	$f3(V_1)$	$f3(f2(f1(b, V_1)))$	$f3(f2(f1(b, V_1)))$	$f3(f2(f1(c, f2(f1(b, V_1)))))$

Table 2: Symbolic Reachability Analysis of N – Inductive Case

ing whether the ϕ_0 or ϕ_1 latches (respectively) are presently transparent. The superscript i means "at time i." For $i > 0$, if $(\Phi^i = 0)$, then $V^i_{C0} = V^i_{B1}$, else $V^i_{C0} = V^{i-1}_{C0}$. Similarly, for $i > 0$, if $(\Phi^i = 0)$, then $V^i_{E1} = V^{i-1}_{E1}$, else $V^i_{E1} = V^i_{D0}$. For the combinational gates, $V^i_{B1} = f1(V^i_{A1}, V^i_{E1})$; $V^i_{D0} = f2(V^i_{C0})$; and finally $V^i_{F1} = f3(V^i_{E1})$.

There are several noteworthy points about this analysis. First, note that the initial values of the ϕ_0 latches are of no semantic importance since the ϕ_0 latches are transparent at time 0. Second, note that the inputs at odd time-steps, denoted with a prime, never propagate. This implies that from each even state, there is exactly one possible next state regardless of input. Finally, the values of the ϕ_1 latches cannot change between an odd and the successive even time-step. For k-phase netlists, similar analysis demonstrates that the initial values of only the ϕ_{k-1} latches propagate (since all others are transparent *before* ϕ_{k-1}), and that primary inputs only propagate at time-steps i such that $i \mod k = 0$ (since inputs combinationally fan out only to ϕ_0, which are transparent only at such time-steps i), and that the ϕ_{k-1} latches stutter from time-steps $k \cdot i - 1, \ldots, k \cdot (i+1) - 2$ (since they are transparent only at time-steps $k \cdot i - 1$). When the ϕ_{k-1} latches are transparent, their values are $\hat{f}\big(I^{k\cdot(i-1)}, (\phi_{k-1})^{k\cdot(i-1)}\big)$, where $\hat{f}$ represents the appropriate composed function.

3.1 Phase-Abstracted Netlists

In this section we propose two abstractions for two-phase netlists. Either may be applied to each MDL of a two-phase netlist independently of the other MDLs, thus yielding an abstraction which has a globally minimum number of state elements (refer to Theorem 5). This minimum would, in general, be less than removing either all ϕ_0 or all ϕ_1 latches. For k-phase designs, we may perform k distinct abstractions per MDL independently of other MDLs by removing all but one phase of latches per MDL.

Either layer of latches of a two-phase netlist N may be removed (and the remaining layer transformed to registers which may be clocked every time-step), and the resulting abstracted netlist will be shown to be bisimilar to the original netlist. Figure 3 shows the first abstraction N' with layer ϕ_1 removed. We label nets equivalently to N to show correspondence of the two netlists. We need a new variable z whose initial value is 1, and thereafter is 0. This variable ensures that the initial value $Z(V_{E1})$ from N is applied to nets V_{E1} in N'.[3] (For k-phase netlists, we introduce z exactly when removing the ϕ_{k-1} latches; otherwise we abstract as per Figure 4.) The initial values of the preserved state variables are $Z(V_{C0})$. For $i > 0$ we have $V^i_{C0} = V^{i-1}_{B1}$. If $(z^i = 1)$, then $V^i_{E1} = Z(V_{E1})^i$, else $V^i_{E1} = V^i_{D0}$. For the other combinational nets, we have that $V^i_{B1} = f1(V^i_{A1}, V^i_{E1})$; $V^i_{D0} = f2(V^i_{C0})$; and $V^i_{F1} = f3(V^i_{E1})$. The symbolic analysis of N' is illustrated in Table 3. The $\uparrow$ indicates that the registers are clocked each time-step. Term $f2^{-1}(V_1)$ refers to an inverse of value V_1 under function $f2$: i.e., $f2^{-1}(V_1) = \alpha$ such that $f2(\alpha) = V_1$. Such an inverse must exist since any value on vector V_{D0} is a combinational function of vector V_{C0}.

Figure 4 illustrates the second abstraction, which removes the ϕ_0 latches. Term $Z(V_{E1})$ is the initial value of the preserved state variables. For $i > 0$, we have $V^i_{E1} = V^{i-1}_{D0}$. For the combinational nets, we have $V^i_{B1} = f1(V^i_{A1}, V^i_{E1})$; $V^i_{C0} = V^i_{B1}$; $V^i_{D0} =$

[3] A similar technique was proposed in [6] for preserving initial values of retimed netlists.

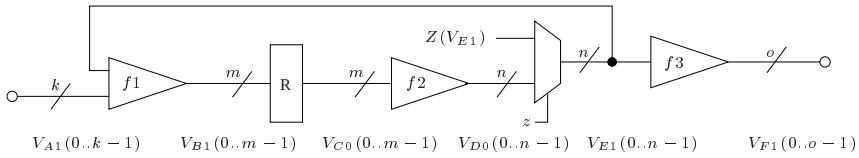

Fig. 3: Abstracted Netlist N'

Net	Time-step 0	...	Time-step j	Time-step $j+1$
Φ	$\uparrow$	...	$\uparrow$	$\uparrow$
V_{A1}	a	...	b	c
V_{B1}	$f1(a, Z(V_{E1}))$	...	$f1(b, V_1)$	$f1(c, f2(f1(b, V_1)))$
V_{C0}	$Z(V_{C0})$	...	$f2^{-1}(V_1)$	$f1(b, V_1)$
V_{D0}	$f2(Z(V_{C0}))$	...	V_1	$f2(f1(b, V_1))$
V_{E1}	$Z(V_{E1})$	...	V_1	$f2(f1(b, V_1))$
V_{F1}	$f3(Z(V_{E1}))$	...	$f3(V_1)$	$f3(f2(f1(b, V_1)))$

Table 3: Symbolic Reachability Analysis of N'

$f2(V^i_{C0})$; and $V^i_{F1} = f3(V^i_{E1})$. Note that the z variable is unnecessary for this abstraction; the initial values of the removed latches do not propagate.

It is noteworthy that either of these two abstractions may be chosen; since the layers may be of differing width $(m \neq n)$, one abstraction may result in a smaller state space than the other. We formally define the phase abstraction process as follows.

Definition 14. A *k-phase abstraction* is a structural transformation of a k-phase (for $k \geq 2$) netlist N to a full-cycle netlist N' such that each structural directed path e from a ϕ_0 to a ϕ_{k-1} latch (including each type exactly once) is transformed as follows.

- Exactly one latch l along e is replaced by a register.
- All latches along e other than l are removed.
- For a ϕ_i latch ($i \neq k-1$) which is removed, it is replaced by a buffer.
- For a ϕ_{k-1} latch which is removed, it is replaced by a multiplexor structure to preserve its initial value as demonstrated by Figure 3.

If l is a ϕ_i latch, the corresponding abstraction is termed a *preserve-ϕ_i* abstraction.

Straightforward analysis demonstrates that valuations of the abstracted ϕ_{k-1}-driven nets at time $i > 0$ are $\hat{f}\left(I^{i-1}, (\phi_{k-1})^{i-1}\right)$, where $\hat{f}$ represents the appropriate composed function.

4 Validity of Abstraction

In this section we define a k-phase bisimulation relation (inspired by Milner's bisimulation relations [16], and related to stuttering bisimulation [9]); this bisimulation is preserved for composition of Moore machines. Further, if two structures are related by a k-phase bisimulation relation, we show that CTL* properties which reason only about *visible vertices* are preserved modulo a simple transformation. We prove a two-phase bisimulation relation for both abstractions presented in Section 3.1.

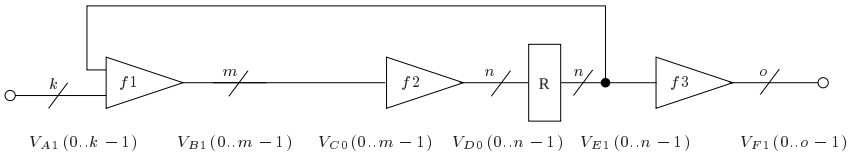

Fig. 4: Alternate Abstracted Netlist N''

Net	Time-step 0	...	Time-step j	Time-step $j+1$
Φ	$\uparrow$	...	$\uparrow$	$\uparrow$
V_{A1}	a	...	b	c
V_{B1}	$f1(a, Z(V_{E1}))$	...	$f1(b, V_1)$	$f1(c, f2(f1(b, V_1)))$
V_{C0}	$f1(a, Z(V_{E1}))$	...	$f1(b, V_1)$	$f1(c, f2(f1(b, V_1)))$
V_{D0}	$f2(f1(a, Z(V_{E1})))$	...	$f2(f1(b, V_1))$	$f2(f1(c, f2(f1(b, V_1))))$
V_{E1}	$Z(V_{E1})$	...	V_1	$f2(f1(b, V_1))$
V_{F1}	$f(Z(V_{E1}))$	...	$f3(V_1)$	$f3(f2(f1(b, V_1)))$

Table 4: Symbolic Reachability Analysis of N''

Definition 15. Let M and M' be structures of a k-phase and a full-cycle netlist N and N', respectively, such that N' was obtained by k-phase abstraction of N. A relation $G \subseteq S \times S'$ is a *k-phase bisimulation* relation if $G(s, s')$ implies:

1. $\mathcal{L}(s) \equiv \mathcal{L}'(s')$.
2. For every $t_1, \ldots, t_k \in S$ such that:
   - $t_1 = s$, and
   - $\bigwedge_{j=1}^{k-1} R(t_j, t_{j+1})$, and
   - $\Phi(t_1) = 0$,

   we have $\bigwedge_{j=1}^{k-2} \mathcal{L}(t_j) = \mathcal{L}(t_{j+1})$, and $\exists t' \in S'$ such that $R'(s', t')$ and $G(t_k, t')$.
3. For every $t' \in S'$ such that $R'(s', t')$, there exist $t_1, \ldots, t_k \in S$ such that:
   - $t_1 = s$, and
   - $\bigwedge_{j=1}^{k-1} R(t_j, t_{j+1})$, and
   - $\Phi(t_1) = 0$, and
   - $\bigwedge_{j=1}^{k-2} \mathcal{L}(t_j) = \mathcal{L}(t_{j+1})$, and
   - $G(t_k, t')$.

We say that a k-phase bisimulation exists from M to M' (denoted by $M \prec M'$) iff there exists a k-phase bisimulation relation G such that for all $s \in S_0$ and $t' \in S'_0$, there exist $t \in S_0$ and $s' \in S'_0$ such that $G(s, s')$ and $G(t, t')$.

We use an equivalence in the above definition since the vertices of M and M' are distinct; however, each visible vertex in M has a unique correspondent in M' as per the phase abstraction algorithm of Definition 14. We require that the two bisimilar machines have corresponding netlists, one being k-phase ($k \geq 2$) and the other being a full-cycle abstraction of the former, not only to emphasize that our structural abstraction algorithm yields a k-phase bisimilar model, but also because without such a requirement our per-MDL abstraction may not yield a k-phase bisimilar composition. As such, note that the k-phase bisimulation relation is neither reflexive, transitive, nor symmetric.

We now prove that our structural transformation yields a bisimilar netlist. An infinite semantic path[4] $\pi = (s_0, s_1, s_2, \ldots)$ is any sequence of states such that any two successive states are related by the transition relation – i.e., $R(s_i, s_{i+1})$. Let π^i denote the suffix path $(s_i, s_{i+1}, s_{i+2}, \ldots)$. We extend the notion of k-phase bisimulation relation to hold between two infinite paths $\pi = (s_0, s_1, s_2, \ldots)$ and $\pi' = (s'_0, s'_1, s'_2, \ldots)$. We will denote this by $G(\pi, \pi')$, which holds iff $G(s_{k \cdot i}, s'_i)$ for every $i \geq 0$. We now state Lemma 2, which follows directly from the above.

Lemma 2. Let s and s' be states of structures M and M', respectively, such that $G(s, s')$. For each infinite path π starting at s and composed of transitions of M, there exists an infinite path π' starting at s' and composed of transitions of M' such that $G(\pi, \pi')$. Similarly, for each infinite path π' starting at s' and composed of transitions of M', there exists an infinite path π starting at s and composed of transitions of M such that $G(\pi, \pi')$.

The set of k-phase-reducible CTL* formulae is a subset of CTL* formulae [17], and is a set of state and path formulae given by the following inductive definition. We also define the k-phase reduction for such formulae. This transformation indicates the relation between the "manual" specification of properties for the abstracted versus the k-phase netlist for Methodology 1, as well as the algorithm for "automatic" translation of properties for Methodology 2.

Definition 16. A *k-phase-reducible (kPR)* CTL* formula φ, and its *k-phase reduction*, denoted by $\Omega(\varphi)$, are defined inductively as follows.

- Every visible atomic proposition p is a kPR state formula $\varphi = p$; $\Omega(\varphi) = p$.
- If p is a kPR state formula, so is $\varphi = \neg p$; $\Omega(\varphi) = \neg\Omega(p)$.
- If p, q are kPR state formulae, so is $\varphi = p \wedge q$; $\Omega(\varphi) = \Omega(p) \wedge \Omega(q)$.
- If p is a kPR path formula, then $\varphi = \mathsf{E}p$ is a kPR state formula; $\Omega(\varphi) = \mathsf{E}\Omega(p)$.
- If p is a kPR path formula, then $\varphi = \mathsf{A}p$ is a kPR state formula; $\Omega(\varphi) = \mathsf{A}\Omega(p)$.
- Each kPR state formula φ is also a kPR path formula φ.
- If p is a kPR path formula, so is $\varphi = \neg p$; $\Omega(\varphi) = \neg\Omega(p)$.
- If p, q are kPR path formulae, so is $\varphi = p \wedge q$; $\Omega(\varphi) = \Omega(p) \wedge \Omega(q)$.
- If p is a kPR path formula, so is $\varphi = \mathsf{X}^k p$; $\Omega(\varphi) = \mathsf{X}\Omega(p)$.
- If p, q are kPR path formulae, so is $\varphi = p\mathsf{U}q$; $\Omega(\varphi) = \Omega(p)\mathsf{U}\Omega(q)$.

Term X^k refers to a series of k X operators. Note that X^k is transformed to X through the reduction; intuitively, this is due to the "multiplying the clock frequency by k," or the replacement of the oscillating clock with an "always active" clock, intrinsic to phase abstraction. As an example, if $\varphi = \mathsf{AG}\big(rdy \rightarrow (\mathsf{AXAX}(req \rightarrow \mathsf{AF}(ack)))\big)$ for a two-phase netlist, then $\Omega(\varphi) = \mathsf{AG}\big(rdy \rightarrow (\mathsf{AX}(req \rightarrow \mathsf{AF}(ack)))\big)$. Note that AXAX is equivalent to AXX; see Section 7.1, page 25 for discussions of AXEX and EXAX. Properties specified over only *visible vertices* may readily be expressed utilizing kPR CTL* since such nets may only toggle every k time-steps; there is no need in practice to express a property with X^i for $i \mod k \neq 0$.

[4] We distinguish a *semantic path*, which is a projection of a *trace* from Definition 2 down to state elements, from a *structural path*. We use the term *path* for the former in the language definitions in keeping with prior research terminology, though reserve this term for the latter elsewhere in this paper in keeping with its structural focus.

Theorem 1. Let s and s' be states of M and M', and $\pi = (s, s_1, s_2, \ldots)$ and $\pi' = (s', s'_1, s'_2, \ldots)$ be infinite paths of M and M', respectively. If G is a k-phase bisimulation relation such that $G(s, s')$ and $G(\pi, \pi')$, then

1. for every k-phase reducible CTL* state formula φ, state $s \models \varphi$ iff $s' \models \Omega(\varphi)$, and
2. for every k-phase reducible CTL* path formula φ, path $\pi \models \varphi$ iff $\pi' \models \Omega(\varphi)$.

Proof. The proof is by induction on the length of formula φ. Our induction hypothesis is that Theorem 1 holds for all kPR CTL* formulae φ' of length $\leq n$.

Base Case: $n = 0$. There are no CTL* formulae of length 0, hence this base case trivially satisfies the induction hypothesis.

Inductive Step: Let φ be an arbitrary kPR CTL* formula of length $n + 1$. We will utilize the induction hypothesis for formulae of length $\leq n$ to prove that Theorem 1 holds for φ. We consider eight cases.

1. If φ is a state formula and an atomic proposition, then $\Omega(\varphi) = \varphi$. Since s and s' share the same labels, state $s \models \varphi \Leftrightarrow s' \models \varphi \Leftrightarrow s' \models \Omega(\varphi)$.
2. If φ is a state formula and $\varphi = \neg\varphi_1$, then $\Omega(\varphi) = \neg\Omega(\varphi_1)$. Using the induction hypothesis (since the length of φ_1 is one less than that of φ), we have that $s \models \varphi_1 \Leftrightarrow s' \models \Omega(\varphi_1)$. Consequently $s \models \varphi \Leftrightarrow s \not\models \varphi_1 \Leftrightarrow s' \not\models \Omega(\varphi_1) \Leftrightarrow s' \models \Omega(\varphi)$.
3. If φ is a state formula and $\varphi = \varphi_1 \wedge \varphi_2$, then $\Omega(\varphi) = \Omega(\varphi_1) \wedge \Omega(\varphi_2)$. By definition, state $s \models \varphi \Leftrightarrow \big((s \models \varphi_1)$ and $(s \models \varphi_2)\big)$. Using the induction hypothesis, since the lengths of φ_1 and φ_2 are strictly less than the length of φ, we have that $s \models \varphi_1 \Leftrightarrow s' \models \Omega(\varphi_1)$, and also that $s \models \varphi_2 \Leftrightarrow s' \models \Omega(\varphi_2)$. Therefore, $s \models \varphi \Leftrightarrow \big(s' \models \Omega(\varphi_1)$ and $s' \models \Omega(\varphi_2)\big) \Leftrightarrow s' \models \Omega(\varphi)$.
4. If $\varphi = \mathsf{E}\varphi_1$, then $\Omega(\varphi) = \mathsf{E}\Omega(\varphi_1)$. The relation $s \models \varphi$ is true iff there exists an infinite path σ beginning at state s such that $\sigma \models \varphi_1$. Consider any σ beginning at s (regardless of whether $\sigma \models \varphi_1$), and let σ' be an infinite path beginning at state s' such that $G(\sigma, \sigma')$. Such a σ' must exist by Lemma 2. Using the induction hypothesis, since the length of φ_1 is exactly one less than the length of φ, we have that $\sigma \models \varphi_1 \Leftrightarrow \sigma' \models \Omega(\varphi_1)$. This implies that $s \models \varphi \Leftrightarrow s' \models \Omega(\varphi)$.
5. If $\varphi = \mathsf{A}\varphi_1$, then $\Omega(\varphi) = \mathsf{A}\Omega(\varphi_1)$. The relation $s \models \varphi$ is true iff for every infinite path σ beginning at state s, we have that $\sigma \models \varphi_1$. For every infinite path σ beginning at s, consider the infinite path σ' beginning at state s' such that $G(\sigma, \sigma')$. Such a σ' must exist by Lemma 2. Applying the induction hypothesis, since the length of φ_1 is exactly one less than the length of φ, we have that $\sigma \models \varphi_1 \Leftrightarrow \sigma' \models \Omega(\varphi_1)$. This implies that $s \models \varphi \Leftrightarrow s' \models \Omega(\varphi)$.
6. Suppose that φ is a path formula which is also a state formula. Since we have exhausted the possibilities for state formulae in the other cases, we conclude that $\pi \models \varphi \Leftrightarrow \pi' \models \Omega(\varphi)$.
7. If $\varphi = \mathsf{X}^k\varphi_1$, then $\Omega(\varphi) = \mathsf{X}\Omega(\varphi_1)$. Note that $\pi \models \varphi$ iff suffix path $\pi^k = (s_k, s_{k+1}, s_{k+2}, \ldots) \models \varphi_1$. Since $G(\pi^k, \pi'^1)$, and using the induction hypothesis (since the length of φ_1 is less than the length of φ), we have that $\pi^k \models \varphi_1 \Leftrightarrow \pi'^1 = (s'_1, s'_2, s'_3, \ldots) \models \Omega(\varphi_1)$. This is equivalent to $\pi \models \varphi \Leftrightarrow \pi' \models \mathsf{X}\Omega(\varphi_1)$, and also to $\pi \models \varphi \Leftrightarrow \pi' \models \Omega(\varphi)$.
8. If $\varphi = \varphi_1 \mathsf{U} \varphi_2$, then $\Omega(\varphi) = \Omega(\varphi_1) \mathsf{U} \Omega(\varphi_2)$. Note that $\pi \models \varphi$ iff there exists i such that $\pi^i \models \varphi_2$ and for all $j < i$, $\pi^j \models \varphi_1$. Since we know that the labeling

of states $s_{k\cdot i-1}, \ldots, s_{k\cdot(i+1)-2}$ are identical, we may without loss of generality assume that i is either 0, or $c \cdot k - 1$ for some c. Using the induction hypothesis, since the lengths of φ_1 and φ_2 are each less than the length of φ, path $\pi^i \models \varphi_2$ iff $\pi'^{i'} \models \Omega(\varphi_2)$ for $i' = (i+1)/k$, rounding down if $i = 0$. Further, path $\pi^j \models \varphi_1$ for all $j < i$ iff $\pi'^{j'} \models \Omega(\varphi_1)$ for all $j' < i'$, implying that $\pi \models \varphi \Leftrightarrow \pi' \models \Omega(\varphi)$.

We now prove k-phase bisimilarity between a k-phase netlist and its k-phase abstraction. Refer to Figures 2-4. Let Φ be the clock variable which alternates between 0 and 1, indicating whether the ϕ_0 or ϕ_1 latches are presently transparent, respectively. The original netlist N has $L^N = V_{C0} \cup V_{E1}$, and $O^N = V_{E1} \cup V_{F1}$. The state space S^N of N is denoted by (c, e, v), comprising of the values of latches V_{C0}, latches V_{E1}, and global clock Φ, respectively. Initial states $S_0^N = \big(Z(V_{C0}), Z(V_{E1}), 0\big)$.

The first abstraction N', which we denote the *preserve*-ϕ_0 abstraction, has $L^{N'} = V_{C0} \cup \{z\}$, and $O^{N'} = V_{E1} \cup V_{F1}$. The state space $S^{N'}$ of N' is denoted by (c, w), comprising the values of V_{C0} and z, respectively. Initial states $S_0^{N'} = \big(Z(V_{C0}), 1\big)$. The second abstraction N'', which we denote the *preserve*-ϕ_1 abstraction, has $L^{N''} = V_{E1}$, and $O^{N''} = V_{E1} \cup V_{F1}$. The state space $S^{N''}$ of N'' is denoted by (e), comprising of the values of latches V_{E1}. Initial states $S_0^{N''} = \big(Z(V_{E1})\big)$. Note that we define O the same in all cases to enable the k-phase bisimulation.

Theorem 2. If N' is a *preserve*-ϕ_0 abstraction of two-phase N, then $N \prec N'$.

Proof. The following relation G between states of N and N' is a two-phase bisimulation relation. Relation G is defined so that holds only for the following two cases.

- $G\big((Z(V_{C0}), Z(V_{E1}), 0), (Z(V_{C0}), 1)\big)$ holds.
- $G\big((c, e, 0), (c, 0)\big)$ holds for any $(c, e, 0)$ reachable from the initial states of N via $k \cdot i$ (for $i > 0$) transitions.

Theorem 3. If N'' is a *preserve*-ϕ_1 abstraction of two-phase N, then $N \prec N''$.

Proof. The following relation G between states of N and N'' is a two-phase bisimulation relation. Relation G is defined so that it holds only for the following two cases.

- $G\big((Z(V_{C0}), Z(V_{E1}), 0), (Z(V_{E1}))\big)$ holds.
- $G\big((c, e, 0), (e)\big)$ holds for any $(c, e, 0)$ reachable from the initial states of N via $k \cdot i$ (for $i > 0$) transitions.

For k-phase netlists, we may generalize any *preserve*-ϕ_{k-1-i} abstraction with proof straightforwardly corresponding to that of Theorem 2 (for $1 \leq i < k$), and a *preserve*-ϕ_{k-1} abstraction with proof corresponding to that of Theorem 3.

Theorem 4. If $M_1 \prec M_1'$ and $M_2 \prec M_2'$, then a k-phase bisimulation exists from the Moore composition $M_1 \parallel M_2$ to the Moore composition $M_1' \parallel M_2'$; equivalently $M_1 \parallel M_2 \prec M_1' \parallel M_2'$.

Proof. This proof is similar to the result that simulation precedence is preserved under Moore composition [15]; we now demonstrate sufficient conditions to extend this result to k-phase bisimilar machines. Note first that M_1 and M_2 are k-phase, while M_1' and M_2' are full-cycle netlists. By the definition of Moore composition, $M_1 \parallel M_2$ is also k-phase and $M_1' \parallel M_2'$ is also full-cycle.

Let (s,t) be an initial state of $M_1 \parallel M_2$. Note that there must exist an initial state (s',t') of $M_1' \parallel M_2'$ with equivalent labels and vice-versa. Any mutual constraints between the initial values of M_1 and M_2 must also exist between M_1' and M_2' – initial value cones must be entirely combinational (before and after composition) so are semantically unaffected by phase abstraction. Additionally, initial values of ϕ_i latches where $i \neq k-1$ are of no semantic importance since they never propagate to a visible vertex. Thus, the two compositions must have label-equivalent initial state sets.

Furthermore, any mutual transition constraints in $M_1 \parallel M_2$ must also exist in $M_1' \parallel M_2'$. For example, if an input of M_1 is driven by an output of M_2 or vice-versa, so too will the corresponding input of M_1' be driven by an output of M_2' or vice-versa. If an input to M_1 is also an input to M_2, so too will the corresponding input to M_1' also be an input to M_2'. Formalizing these facts, for each $(s_i,t_i) \in S_0^{M_1\|M_2}$ and $(s_i',t_i') \in S_0^{M_1'\|M_2'}$, there exist $(s_j,t_j) \in S_0^{M_1\|M_2}$ and $(s_j',t_j') \in S_0^{M_1'\|M_2'}$ such that $G\big((s_i,t_i),(s_j',t_j')\big)$ and $G\big((s_j,t_j),(s_i',t_i')\big)$, which satisfies Definition 15.

Theorems 1-4 allow us to apply the various abstractions independently on each dependent layer, and still preserve model checking of kPR CTL* formulae on the composition of the abstractions.

Corollary 1. If N''' is obtained from N by applying any *preserve-ϕ_i* abstraction independently on each of its minimal dependent layers, then $N \prec N'''$.

4.1 k-Phase Bisimulation versus Stuttering Bisimulation

As stated earlier, there are significant similarities between k-phase bisimulation and stuttering bisimulation. In particular, a k-phase netlist N and its abstraction N' are related by each state s' of the state transition graph of N' being replaced by k states $s_1,\ldots,s_k$ in N with identical labels as s', such that each transition (t',s') in N' has a corresponding state transition (t_k,s_1) in N (and vice versa), and each transition (s',t') in N' has a corresponding state transition (s_k,t_1) (and vice versa), and (s_i,s_{i+1}) for $1 \leq i < k$ are the only other transitions involving each s_i. Stuttering bisimilarity would relate each s' to each s_i; indeed, the results of prior research on stuttering bisimulation hold between N and N', which accounts for many of the results of Section 4.

There are several important distinctions between these topics. First, note that unlike stuttering bisimulation, we do preserve CTL* formulae over X^k. This is because we relate each s' only to s_1. This may practically be circumvented by adding a clock Φ to N, and instead of using the X operator as is disallowed by stuttering bisimilarity, we use the generalized $\mathsf{next\_event}(a)(f)$ operator [18] which means that "f must hold the next time that a holds," and defining a to hold exactly when $\Phi = 0$.

However, there is one fundamental contribution of this work beyond (or leading to) stuttering bisimulation: we provide linear-time algorithms that analyze and transform

Algorithm Phase_Abstract($\tilde{N}$)

1. Color $\tilde{N}$, to yield k-phase netlist N.
2. If $k < 2$, or if $\tilde{N}$ cannot be consistently colored, no abstraction is possible; return *unsuitable*.
3. Partition N into MDLs: $Q_0, \ldots, Q_n =$ **Partition(**N**)**.
4. For each MDL Q_i:
   (a) Let $j \in [0, k)$ be the largest number such that $\forall a \in [0, k).(a \neq j) \rightarrow (|\phi_a \cap Q_i| \geq |\phi_j \cap Q_i|)$.
   (b) Perform the chosen *preserve*-ϕ_j abstraction on Q_i.
5. Return abstracted composite netlist N'''.

Fig. 5: Pseudo-code for Phase Abstraction Algorithm

the structure of a netlist, hence there is no need to transform or even analyze the state transition graph of a netlist to achieve our reductions. Thus, while stuttering bisimulation is a good framework from which to theoretically understand phase abstraction, it does not offer a practically useful mechanism to perform phase abstraction on large netlists. Furthermore, note that we restrict our k-phase bisimilarity from Definition 15 to hold only between the structures of corresponding k-phase and phase abstracted netlists. This is partially a simple mechanism to enable use of X^k – which is an important operator in hardware verification, and also a mechanism to allow us to partition and independently abstract each MDL of a k-phase netlist and still yield a k-phase bisimilar composition. Note that an arbitrary number of structures may be stuttering bisimilar, but their respective compositions not be bisimilar, for example, due to differing joint transition constraints. Restricting ourselves to k-phase and corresponding phase abstracted netlists overcomes this limitation.

5 Algorithms for the Abstraction of k-Phase Netlists

In this section we discuss our algorithms for abstracting k-phase netlists. We utilize a simple linear-time algorithm to color the netlist in the first step of **Phase_Abstract()** shown in Figure 5. The algorithm merely exercises the coloring formalized in Definition 8 relative to a seed. The seed may be readily obtained in several ways: e.g., relative to the primary inputs and outputs, or relative to conventions in the clocking logic. After a partitioning of the colored netlist into MDLs via algorithm **Partition()** shown in Figure 6, we choose which abstraction to perform on each MDL, then perform the abstraction as per Definition 14.

Note that the algorithm may readily be optimized so that each net is considered only a constant number of times in fanout traversal as well as fanin traversal. We thereby ensure that, like a breadth-first search, its asymptotic complexity is $O(|vertices| + |edges|)$. This linearity ensures that it will consume negligible resources for even the largest designs we have considered for model checking (more than 8,000 latches).

Theorem 5. Algorithm **Phase_Abstract()** yields optimal k-phase abstraction reductions for two-phase netlists.

Algorithm Partition(N)

$i = -1$;
for each latch $v \in L$ {
  if ($v \in \bigcup_{j=0}^{i} Q_j$) {continue;}
  i++;
  Q_i = in_q = out_q = $\{v\}$;
  while $\big(\neg empty(\text{in\_q}) \vee \neg empty(\text{out\_q})\big)$ {
    if ($\neg empty(\text{in\_q})$) {
      $u = pop(\text{in\_q})$;
      if $\big(C(u) \equiv 0\big)$ {continue;}
      $\beta = \{w : w \in$ *combinational fanin*$\big(data(u)\big) \wedge w \in L\}$;
      $assert\big(C(\beta) \equiv C(u) - 1\big)$;
      $\beta = \beta \setminus Q_i$;
      $push(\text{in\_q}, \beta)$; $push(\text{out\_q}, \beta)$;
      $Q_i = Q_i \cup \beta$;
    }
    if $\big(\neg empty(\text{out\_q})\big)$ {
      $u = pop(\text{out\_q})$;
      if$\big(C(u) \equiv k - 1\big)$ {continue;}
      $\beta = \{w : w \in$ *combinational fanout*$(u) \wedge w \in L\}$;
      $assert\big(C(\beta) \equiv C(u) + 1\big)$;
      $\beta = \beta \setminus Q_i$;
      $push(\text{in\_q}, \beta)$; $push(\text{out\_q}, \beta)$;
      $Q_i = Q_i \cup \beta$;
    }
  }
}
Return $Q_0, \ldots, Q_i$;

Fig. 6: Pseudo-code for MDL Partitioning Algorithm

Proof. By Lemma 1, there is a unique MDL partition of a netlist. Each MDL is of minimum size, resulting in a maximum number of dependent layers in the netlist. Note that along any shortest input-output path within a single MDL, exactly one register must exist after abstraction – if zero or more than one exist, a k-phase bisimulation is clearly broken. This observation provides the justification for partitioning into MDLs. Since each MDL may be abstracted independently of the others, the locally optimal solutions yield a globally optimal result for two-phase netlists. This property follows from the observation that eliminating any latch l from a two-phase MDL implies that each latch l' within the MDL in the combinational fanin of $data(l)$, or in the combinational fanout of l, must be preserved. This in turn implies that each latch within the MDL in the combinational fanin of $data(l')$, or in the combinational fanout of l', must be eliminated, and so on.

For the uncommon case of k-phase designs with $k \geq 3$, this linear-time algorithm may not yield an optimal solution. For example, consider the 3-phase MDL of Figure 7, where the numbers indicate the "width" of the corresponding vectored latches. Preserving any single phase will not yield an optimal solution; preserving the ϕ_1 B and the

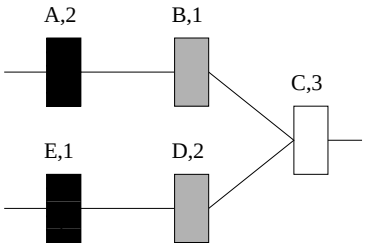

Fig. 7: Three-Phase MDL

$\phi_0\ E$ yields an optimum solution of two (versus a solution of three that the linear-time algorithm would produce). Clearly the optimum solution is achievable in superlinear polynomial time by solving a s-t node min-cut problem[5] on the latch connectivity graph (whose vertices are all latches, whose directed edges represent a combinational fanout connectivity between latches of color i to color $i+1$ for $0 \leq i < k-1$) between *sources* of color 0 and *sinks* of color $k-1$. Rather than spending superlinear resources for phase abstraction, it is our experience that an optimal tool implementation first uses a linear technique for efficient phase abstraction (while obtaining superior reductions to a global *preserve-*ϕ_i approach [4]), then subsequently uses other more aggressive superlinear techniques "under the covers" such as retiming [2] for additional reductions.

6 Experimental Results

Methodology 1 of our two-phase abstraction algorithm has been implemented in *RuleBase* [3], the model checker developed by IBM Haifa Research Laboratory as an extension to SMV [20]. It is deployed as a first-pass netlist reduction technique; the phase abstracted netlist is saved and used as the basis for further optimizations before being model checked. This algorithm has been deployed upon many components of IBM's Gigahertz Processor. The results of this reduction on several components of this processor are provided in Table 5. These numbers do not reflect the results of any other reduction techniques. During the initial stages of model checking, this abstraction was not available. Once the abstraction became available, properties which previously required many hours to complete would finish in several minutes. More encompassing properties became feasible on the abstracted netlist which would not otherwise complete.

Our experiments were run on an IBM RS/6000 Workstation Model 590 with 2 GB main memory. We utilized the most rigorous automatic model reduction techniques RuleBase has to offer. These experiments were run with a random initial BDD order (though pairing present-state and next-state variables for each state element), and with dynamic reordering enabled using the technique of Rudell [21]. The reasoning behind using a random initial order includes the following points.

– At the initial stages of a verification effort, no orders are available.

[5] One of the most efficient known algorithms for solving the s-t min-cut problem is the *highest-label preflow-push algorithm*, which is $O(|vertices|^2 \cdot |edges|^{1/2})$ [19].

Logic Function	State Bits Before Reduction	State Bits After Reduction
Load Serialization Logic	8096	2586
L1 Cache Reload Logic	3102	1418
Instruction Flushing Logic	138	69
Instruction-Fetch Address Generation Logic	4891	2196
Branch Logic	6918	3290
Instruction Issue Logic	6578	3249
Tag Management Logic	578	289
Instruction Decode Logic	1980	978
Load / Store Control	821	409

Table 5: Two-Phase Reduction Results

– These results capture the difficulty of calculating a reasonable BDD order before and after abstraction (since reordering times are included in both results).

One property run on the Load Serialization Logic which required 25.6 seconds, 36 MB of memory on the abstracted netlist (with 81 variables) required 450.2 seconds, 92 MB of memory for the unabstracted netlist (with 116 variables). This time includes that necessary to perform phase abstraction. This inclusion is somewhat unfair, since the netlist may be phase abstracted and saved upon HDL compilation for arbitrary reuse. However, this abstraction time is negligible even on large netlists.

A more challenging property for the Instruction Flushing Logic required 852 seconds of user time, 48 MB on the abstracted netlist (with 96 variables). This same property did not complete on the unabstracted netlist (with 162 variables) within 72 hours.

While it may seem surprising in the above two examples that (after other reduction techniques) the number of variables after phase abstraction is *more than half* that without phase abstraction, this is due to two phenomena. First, some of these variables are used for environment and properties; these may be modeled directly as registers (rather than ϕ_0-ϕ_1 latches) even for the unabstracted two-phase netlist. Second, in some cases, RuleBase was able to exploit some redundancy among the level-sensitive latches through other model reduction techniques (e.g., variable equivalence).

Due to the linearity of this algorithm, and the speedups that it may offer to other more expensive reduction techniques (e.g., retiming and structural simplification), we recommend that it be used as a first pass reduction either before or after a linear-time cone-of-influence reduction. The benefits of this algorithm extend beyond the reduction in state depth, which reduces the time and memory consumed by reachability analysis. BDD reordering time is often greatly reduced since the BDDs tend to be significantly smaller, and since with less variables a "good order" tends to be faster to compute. Phase abstraction also reduces the number of image calculations necessary to reach a fixed-point or on-the-fly failure since the diameter of the machine is divided by k. Further, the removal of latches allows "collapsing" of adjacent functions to a single function, which may be represented efficiently on the same BDD and which increases the domain of applicability of combinational simplification techniques. However, this also risks explosion of BDD size; advanced techniques such as implicit conjoining [22],

or fine-grained reachability analysis [23], may be used to minimize such risk. Nevertheless, as our experiments demonstrate, this abstraction tends to enhance BDD-based analysis. We have not observed one case where phase abstraction hurt a verification effort to the point where it needed to be disabled during five years of using this technique. An efficient algorithm for image calculation upon k-phase netlists is presented in [4], though it does not offer the above-mentioned benefits of our structural transformation.

As demonstrated, this abstraction enabled model checking to verify more encompassing properties in less time. Users of this technique often find that writing properties and environments for the phase abstracted netlists is more intuitive than for the corresponding two-phase netlists. All RuleBase users quickly converted to running exclusively with this abstraction. There have been more than one thousand formulae written and model checked to date on this project, which collectively have exposed more than two hundred bugs at various design stages. This algorithm thus provided an efficient and necessary means by which to free ourselves from the verification burdens imposed by the low level of the high-performance implementation.

It is noteworthy that approximately 150 HDL bugs were found by this algorithm due to violations of ϕ_0-ϕ_1 connectivity. While synthesis tools report such timing violations, the benefits of this abstraction provided strong motivation to correct these errors earlier in the design phase. Our algorithms tended to be computationally more efficient than the timing analysis algorithms due to their operation on more abstract behavioral models, hence our algorithms began to be used in cases merely to quickly identify such flaws. Many simulation and model checking frameworks treat such logic "properly" for unphase-abstracted verification even if such connectivity problems exist – these platforms often assume zero combinational delay, but no combinational "flow-through" for two adjacent level-sensitive latches even if both are simultaneously transparent.

7 Implementation Details

In this section we discuss details relating to the implementation and deployment of phase abstraction. We first discuss a netlist preprocessing technique to increase the fragment of netlists which may be phase abstracted. We next discuss trace generation details. We lastly discuss some implications of the limited expressibility of k-phase reducible CTL* formulae.

7.1 Netlist Preprocessing

As per Definitions 8 and 9, the inputs of k-phase netlists combinationally fan out only to ϕ_0 latches, and their outputs have only ϕ_{k-1} latches in their combinational fanin, which may be unnecessarily limiting in practice. We discuss implications of *invariant checking* simple properties of the form $\mathsf{AG}(v)$ for arbitrary vertices v in [24], which is sound and complete for any color $C(v)$, even when v has inputs in its combinational fanin. For model checking, we choose to treat inputs which combinationally fan out to ϕ_i latches (for $i \neq 0$) as if they passed through an entire flow of $\phi_0, \ldots, \phi_{k-1}$ latches, since designers tend to view such inputs as arriving "late" in the corresponding clock period. Therefore, we wish to prevent the algorithm from eliminating all latches

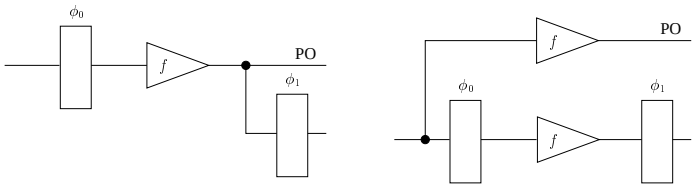

Fig. 8: Aligning ϕ_0-Driven Outputs

$\phi_i, \ldots, \phi_{k-1}$ if any input combinationally fans out to ϕ_i. We handle such inputs by introducing "dummy" pipelined $\phi_0, \ldots, \phi_{i-1}$ latches between such inputs and their sinks; their initial values are unimportant since they will never be visible.

To ensure compositionality of specification, we remove any $\phi_0, \ldots, \phi_{k-2}$ latches which combinationally fan out to primary outputs. Our technique replicates the combinational logic in the fanin cone of such outputs within the MDL, and connects the replicated cones to the corresponding sources of the ϕ_0 latches, thereby effectively removing the offending latches. This is depicted in Figure 8 for a two-phase netlist; the netlist to the left is the original netlist, and to the right the netlist after preprocessing.

One additional preprocessing is for netlists with gated clocks, which would violate Definition 8. Transformation of a netlist with gated clocks to one without gated clocks is a straightforward automatic process. If the clock of a given latch l is driven by a conjunction of a *gate* signal with a free-running clock, we remove the AND vertex from the clock, and utilize the *gate* signal to select a newly-introduced multiplexor which will drive the data input to l. The "*gate* open" multiplexor input is connected to the original data source of l; the "*gate* closed" input is connected to l', a latched version of l with the same initial value as l (which we may need to create). The clock to the latch is next connected to its free-running version. This transformation is shown in Figure 9 for a two-phase netlist; the netlist to the left has a gated clock, and the netlist to the right is the functionally-equivalent version with no gated clock. *CLK_* refers to a free-running clock with phase opposite to *CLK*.

7.2 Trace Generation

Since we verify the phase-abstracted netlist, any witness or counterexample traces will be in terms of the abstracted netlist. While users of Methodology 1 often find such

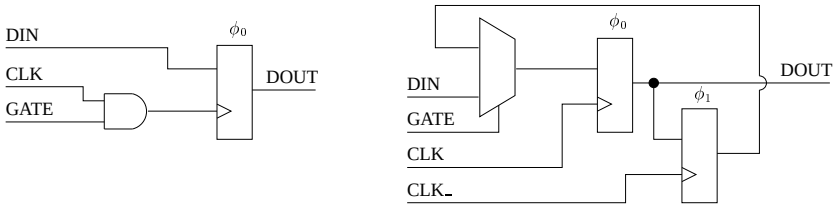

Fig. 9: Removing Clock Gates

Algorithm Lift_Trace($p, N, \hat{Q}$)

1. For each $v \in V$, define $\epsilon(v) = 1$ if the MDL containing v was abstracted via a *preserve-ϕ_j* abstraction for $j > C(v)$, else $\epsilon(v) = 0$.
2. Form the un-phase-abstracted trace p' as follows. For each v_p^i (where v is not an element of the clock logic D), set $v_{p'}^j = v_p^i$ for every $j \in \{\{k \cdot \big(i + \epsilon(v) - 1\big) + C(v), \ldots, k \cdot \big(i + \epsilon(v)\big) + C(v) - 1\} \setminus \{c : c < 0\} \setminus \{c : c < C(v) \wedge C(v) \neq k - 1\}\}$.
3. Return p'.

Fig. 10: Pseudo-code for Trace Lifting Algorithm

abstracted traces easier to view, occasionally it is desired to undo the effects of phase abstraction in the traces. Our algorithm for lifting (i.e., *un-phase-abstracting*) a trace requires the phase abstracted trace, the k-phase netlist N, and the MDL partition with the corresponding data as to which abstraction was performed per partition, denoted $\hat{Q}$, and is provided in Figure 10.

The trace-lifting algorithm neglects populating values into p' for negative time-steps since time is defined only on $\mathbb{N}$. Note that we do not populate values into the trace for color $c \in \{1, \ldots, k-2\}$ vertices for time-steps $0, \ldots, c-1$; this is to prevent the initial values of latches of color $i \neq k-1$ from becoming visible in the un-phase-abstracted trace. Note that color-$k-1$ vertices are properly temporally aligned as per the nature of our abstraction, so we merely repeat their values in the phase-abstracted trace from time i to times $k \cdot i - 1, \ldots, k \cdot (i+1) - 2$ in p'. The other colors may or may not be aligned depending on which abstraction was performed on their MDL. Term $\epsilon(v)$ captures the type of abstraction; any valuation to a color j vertex in a MDL which was abstracted with a *preserve-ϕ_{j+1}* or "greater" abstraction is early and must be pushed forward (thus increments the multiple of k). The addition of $C(v)$ is responsible for rooting the valuations to the proper offset within the clock period. And, of course, the values are repeated for a total of k time-steps. Running this algorithm against the example netlist in Section 3 will convince the reader that it works for $k = 2$; extending this example to a 3-phase netlist will offer additional insight into its mechanics.

7.3 kPR CTL* Expressibility

Our technique is applicable only to netlists for which specification reasons solely about color-$k-1$ nets. Using a dual of the analysis presented in this paper, our technique is also applicable to the case that the specification refers only to color-i (for $i < k-1$) nets. If specification refers concurrently to multiple colors of nets, Methodology 2 is often preferred. Alternatively, we may force the tool to preserve a given (user-known) color of latches by restricting its freedom to choose which type to preserve per MDL, to allow the user to create a consistent specification using Methodology 1. We do not provide detailed analysis of these variants in this paper, but they are straightforward.

We require that properties do not reason about the clock logic. With a two-phase design, one may for example utilize the clause $(clock = 0) \wedge p$ to bind p to an even time-step. After our abstraction, the notion of even and odd time-steps disappears, and

through the process of phase abstraction the clock logic is discarded (and the resulting registers are implicitly clocked every time-step).

Lastly, consider the CTL* formulae of the form $\mathsf{AXEX}(p)$ and $\mathsf{EXAX}(p)$. As illustrated by Tables 1 and 2, every i-th state (for $i \mod k \neq k-1$) of a k-phase netlist has only one possible next state since inputs are ignored at those times. For a two-phase netlist, if the clause $\mathsf{AXEX}(p)$ is bound to an even state, it is equivalent to $\mathsf{EXX}(p)$; the AX would be equivalent to an EX relative to the even state. Similarly, $\mathsf{EXAX}(p)$ bound to an even state is equivalent to $\mathsf{AXX}(p)$. We have not encountered a single property which we wished to verify which required such a clause, hence argue that these formulae are rare if not useless in the practical verification of k-phase netlists. Our formalisms disallow verification of such properties through phase abstraction.

8 Conclusions

In this paper we have developed efficient linear-time algorithms for identifying and abstracting k-phase netlists for enhanced verification. The benefits of our phase abstraction include much smaller verification time and memory requirements through "shallower" state depth – often less than $1/k$ of that necessary without the abstraction which reduces complexity of the transition relation and simplifies BDD reordering, as well as a reduction of machine diameter by a factor of $1/k$. Additionally, our technique allows a user to define properties and environments more abstractly. We establish a bisimulation relation between the original and phase-abstracted netlists. Our linear-time reduction is optimal for two-phase designs (and near-optimal for $k > 2$), and is valid for model checking CTL* formulae which reason solely about latches of a given phase. Experimental results from the deployment of this algorithm (as implemented in the model checker RuleBase) upon IBM's Gigahertz Processor are provided, and illustrate its extreme practical benefit.

References

1. N. Weste, K. Eshraghian, and M. J. S. Smith. *Principles of CMOS VLSI Design: A Systems Perspective*. Addison Wesley, 2001.
2. A. Kuehlmann and J. Baumgartner. Transformation-Based Verification using Generalized Retiming. In *Proceedings of the Conference on Computer-Aided Verification*, pp. 104-117, July 2001.
3. I. Beer, S. Ben-David, C. Eisner, and A. Landver. RuleBase: an Industry-Oriented Formal Verification Tool. In *Proceedings of the Design Automation Conference*, pp. 655-660, June 1996.
4. G. Hasteer, A. Mathur, and P. Banerjee. Efficient Equivalence Checking of Multi-Phase Designs Using Phase Abstraction and Retiming. In *ACM Transactions on Design Automation of Electronic Systems*, pp. 600-625, October 1998.
5. C. E. Leiserson and J. B. Saxe. Optimizing Synchronous Systems. In *Journal of VLSI and Computer Systems*, 1(1):41–67, January 1983.
6. H. Touati and R. Brayton. Computing the Initial States of Retimed Circuits. In *IEEE Transactions on Computer-Aided Design*, 12(1):157–162, January 1993.

7. C. E. Leiserson and J. B. Saxe. Retiming Synchronous Circuitry. In *Algorithmica*, 6(1):5–35, 1991.
8. J. B. Orlin. A Faster Strongly Polynomial Minimum Cost Flow Algorithm. In *Proceedings of the 20th ACM Symposium on the Theory of Computing*, pp. 377-387, May 1988.
9. M. C. Browne, E. M. Clarke, and O. Grumberg. Characterizing Finite Kripke Structures in Propositional Temporal Logic. *Theoretical Computer Science*, 59:115-131, 1988.
10. E. A. Emerson and J. Y. Halpern. "Sometimes" and "Not Never" Revisited: On Branching Time versus Linear Time Temporal Logic. *Journal of the ACM*, 33(1):151-178, 1986.
11. J. Baumgartner, A. Tripp, A. Aziz, V. Singhal, and F. Andersen. An Abstraction Algorithm for the Verification of Generalized C-Slow Designs. In *Proceedings of the Conference on Computer-Aided Verification*, pp. 5-19, July 2000.
12. T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Assume-Guarantee Refinement Between Different Time Scales. In *Proceedings of the Conference on Computer-Aided Verification*, pp. 208-221, July 1999.
13. A. Albrecht and A. J. Hu. Register Transformations with Multiple Clock Domains. In *Proceedings of the Conference on Correct Hardware Design and Verification Methods*, pp. 126-139, September 2001.
14. Z. Kohavi. *Switching and Finite Automata Theory*. Computer Science Series. McGraw-Hill Book Company, 1970.
15. O. Grumberg and D. E. Long. Module Checking and Modular Verification. In *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.
16. R. Milner. *Communication and Concurrency*. Prentice Hall, New York, 1989.
17. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. In *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.
18. I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The Temporal Logic *Sugar*. In *Proceedings of the Conference on Computer-Aided Verification*, pp. 363-367, July 2001.
19. J. Cheriyan and S. N. Maheshwari. Analysis of Preflow Push Algorithms for Maximum Network Flow. *SIAM Journal on Computing*, 18:1057-1086, 1989.
20. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
21. R. Rudell. Dynamic Variable Ordering for Ordered Binary Decision Diagrams. In *IEEE/ACM International Conference on Computer-Aided Design*, pp. 42–47, November 1993.
22. A. J. Hu, G. York, and D. L. Dill. New Techniques for Efficient Verification with Implicitly Conjoined BDDs. In *Proceedings of the Design Automation Conference*, pp. 276-282, June 1994.
23. H. Jin, A. Kuehlmann, and F. Somenzi. Fine-Grain Conjunction Scheduling for Symbolic Reachability Analysis. *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 312–326, April 2002.
24. J. Baumgartner. Automatic Structural Abstraction Techniques for Enhanced Verification. PhD Thesis, University of Texas at Austin, December 2002.