Automatic Generalized Phase Abstraction for Formal Verification

Per Bjesse and James Kukula
Synopsys Inc.

***Abstract*— A standard approach to improving circuit performance is to use an N-phase design style where combinational logic is interspersed freely between level sensitive latches controlled by separate clocks. Unfortunately, the use of an N-phase design style will increase the number of state variables by a factor of N, making formal verification many orders of magnitude harder. Previous approaches to solving this problem restrict the kind of designs that can be handled severely and construct an abstracted netlist with fewer state variables by a syntactic analysis that requires the user to identify clocks. We extend the current state of the art by introducing a phase abstraction algorithm that (1) poses no restrictions on the design style that can be used, that (2) avoids an error prone syntactic analysis, that (3) requires no input from users, and that (4) can be integrated into any model checker without requiring HDL code analysis.**

I. Introduction

There are many approaches possible for proving that a design functions correctly. Ultimately they all rely on analysis methods that are very sensitive to the number of gates and registers in the formal model. For example, hard properties are often resolved using some form of abstraction coupled with fixpoint computation using Binary Decision Diagrams (BDDs). It is well-known that BDDs can only reliably cope with at most a hundred state bits or so in the abstraction of the design under analysis. Moreover, many other algorithms for exploring the state space of a circuit use satisfiability checking—an analysis whose complexity is critically dependent on the amount of logic in the circuit that is analyzed.

Today's high performance design styles often complicate an already hard verification problem by introducing extra gates and registers that are functionally redundant but allow the circuit implementation to meet stringent performance constraints. One such design style is *multi-phase clocking*, where several state elements, each driven by a separate clock, implement a single virtual state bit. While this increases robustness and allows relaxed timing constraints, it complicates formal verification considerably: In an N-phase design on the order of N latches are used in place of every register. A budget of a hundred state bits or so is hence likely to be eaten up quickly. Moreover, multiphase clocking is used pervasively in the industry as a performance improvement technique across whole designs, so the state multiplication effect is not localized. It is hence important to find some way to mitigate the increased verification complexity.

As an example of multi-phase clocking, consider the design style schematically represented in Figure 1. In this picture, triangles denote combinational logic and rectangles denote banks of state-holding elements. In the remainder of this paper we will refer to this form of design as the *rigid two-phase* design style.

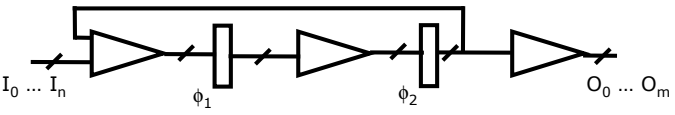

Fig. 1. Rigid two-phase design style.

In the rigid two-phase design style there are two types of state-holding elements, ϕ_1 latches and ϕ_2 latches, each fed by a clock that does not overlap with the clock feeding the other latch type. ϕ_1 latches can only depend on inputs and ϕ_2 latches, whereas ϕ_2 latches can only depend on ϕ_1 elements. The outputs of a strict two-phase design is only allowed to depend on ϕ_2 elements.

Since a given multi-phase design is an implementation of an ideal simpler single phase design, a natural approach to improving the complexity behavior of our formal verification algorithms is to attempt to extract an equivalent "full cycle" model from the multi-phase design under analysis. The process of deriving this simpler design is commonly referred to as *phase abstraction* ([1], [2], [3]).

In the case of a strict two-phase design style, it is not hard to show that the validity status of safety properties will be preserved if the phase abstracted system is generated by substituting either a short circuit or a multiplexor for one of the latch layers, and flip flops for the other [2]. If there are as many ϕ_1 latches as ϕ_2 latches, this means that the number of state-holding elements in the abstracted design is reduced by 50%.

The standard approach to performing two-phase abstraction for model checking proceeds as follows [2]:

1) The user identifies the points where the non-overlapping clocks that drive the different layers of latches are generated or injected, and their phase number.
2) The system propagates this information to identify as many latches as possible as ϕ_1 or ϕ_2 elements.
3) Given this information, regions where a strict two-phase design style is followed are identified.
4) For each such region, or *Minimal Dependent Layer* (MDL) in the terminology of [2], either all the ϕ_1 or ϕ_2 latches are removed by the appropriate substitution while the other latches become flip flops.

0-7803-9254-X/05/$20.00 ©2005 IEEE.

It is clear that phase abstraction can be a very powerful abstraction technique. However, if it is going to be used outside a controlled environment, there is a number of obstacles that have to be overcome. First of all, in the standard approach the user has to identify the different clocks and their phase relationship. In a commercial formal verification tool, this is a serious weakness—each new piece of data the user has to enter increases the threshold to widespread adoption, and allows room for human error. Second, standard two-phase abstraction is restricted to a very strict design style. This does not pose a problem if we can control the design methodology. However, a tool aiming at a heterogeneous user group should assume as little as possible about design rules.

Particular problems with the strict design style include the following:

1) What if outputs have both ϕ_1 and ϕ_2 latches in their support?
2) What if the design freely mixes ϕ_1 and ϕ_2 latches, without following the strict two phase rules?
3) What if the design contains flip-flops freely interspersed with latches?
4) What if clocks overlap?
5) What if clocks are gated?

Issues 1, 2, and 3 are obstacles that will separate a larger group of logic and state elements into much smaller MDLs, thus reducing the potential for subsequent reductions. This is bad by itself. However, the show stoppers in many cases are issues 4 and 5. Item 4 will cause trouble with a number of systems, especially if there are multiple independently clocked regions or more esoteric clocking schemes present. Item 5 is a symptom of a much larger problem: The standard analysis is completely syntactic in that it hinges on a pure graph theoretic connectivity analysis that uses no semantics for the system under analysis.

In this paper, we present a new approach to phase abstraction. Our analysis does not require any rules to be followed in the design for a good result, and it does not require any extra user input. In particular, it has the following advantages over the standard approach:

- We do not require the user to identify clock generation networks or phase relationships. However, if the user chooses to provide some clocking information, we can still make use of it.
- Our analysis does not require the user to follow any special design rules. We allow outputs fed by arbitrary state elements and we pose no restriction on how inputs can be read.
- We allow clocks that overlap, and that have arbitrary mutual relationships.
- We allow arbitrary logic in clock networks.

Moreover, our analysis is not tied to standard k-phase clocking; any arbitrary mix of different clocking schemes is allowed.

Our analysis proceed in several steps. In the first pass, we apply an automatic analysis to find as many clock generation networks as possible. This information is then analyzed to compute a probable number of phases to extract. Given the computed clock information and the number of phases to abstract, we transform the original system model into a new design whose correctness we can relate to the correctness of the original system.

II. Preliminaries

In the remainder of the presentation, we will assume that the initial representation that we start with is the standard implicitly clocked synchronous model that comes out of the front-end of a model checking tool. We thus assume that all the standard semantics-preserving transformations such as inclusion of environment models, clock modeling, latch modeling, resolution of combinational loops, and resolution of multiply driven nets have been done. The resulting model uses simple stateholding elements with a single input and a single output, triggered by an implicit global clock—level sensitive behavior has been remodeled using bypass/hold logic, and clock drivers have been folded into the data lines. This model is *complete*, in the sense that no extraneous logic is necessary for determining the status of properties. We also assume that we have been given a computed reset state `1`,`0`, or `X` for all state-holding elements in our model. Furthermore, as the focus of this paper is on the checking of safety properties, we assume (without loss of generality) that we are interested in checking whether a particular output O of our design is stuck at one.

These assumptions all hold for the problem descriptions generated internally in our commercial model checker (and all other commercial model checkers that we are aware of), so we do not believe them overly restrictive.

III. Identification of internal clock generation networks

As our representation of the design under analysis is a complete model, all the different clocks that are used to control the update of state elements are generated somewhere in the netlist under analysis. Our first task is to find the places where derived clocks are generated. We will do this by attempting to locate places in the design where some bit pattern will be repeated indefinitely.

In order to get phase abstraction to work well, the method used to find derived clocks must fulfill two criteria: First of all, it must be very fast so as not to slow down the model construction. In order to achieve this, we will opt for a conservative but fast heuristic (failing to identify a clock in our framework may impact size of the abstracted model, but cannot affect the correctness of our abstraction). Second, the method must generate enough information to make our analysis useful. In our experience, the following algorithm represents a good tradeoff between these criteria.

Let us make some definitions. A signal is said to be *clock-like* if it can be generated by repeating some boolean signal pattern, a *generator*, over and over again. Generators are not unique—the generators $[1,0]^*$ and $[1,0,1,0]^*$ both generate the clock-like signal $1,0,1,0\ldots$. Every clock-like signal has an associated *minperiod*, which is the length of its shortest

generators. The focus of our clock identification strategy is to find as many state elements as possible that are clock-like, and compute their minperiod generators.

In order to keep computation time to a minimum, we use three valued simulation to identify clock-like signals. Our *cycle identification algorithm* proceeds as follows:

1) `Visited` $=$ [`ResetState`], $s =$ `ResetState`
2) $s' = Sim_X(s)$
3) If $s' \notin$ `Visited` then append s' to `Visited`, assign s' to s, and goto 2. Otherwise partition `Visited` into a *stem* and a *cycle* part by splitting the sequence just before the previous copy of s', and then terminate.

In the algorithm we make use of the function $Sim_X()$ that computes the three valued next state resulting from the application of `X`s to each input from the given state. We also use a list `Visited` to maintain the states seen in the three valued simulation run so far.

The cycle identification algorithm necessarily terminates, as the number of possible simulation steps is bounded by $3^{|\texttt{Regs}|}$ if we have $|$`Regs`$|$ state-holding elements. While this is a very large theoretical bound, we terminate quickly in practice as the injection of `X`s on all inputs in each simulation step will make most state bits go to `X` quickly.

What information can we glean from the generated stem and cycle information? Since the algorithm terminates when we have reached a new state S_m that is equal to some previous state S_n we know that S_{m+1} must be equal to S_{n+1} as $S_{m+1} = Sim_X(S_m) = Sim_X(S_n) = S_{n+1}$. By repeated application of the same reasoning, it is easy to see that the infinite state sequence that would be generated by iterated application of Sim_X to `ResetState` can be formed by concatenating the stem with infinitely many copies of the cycle.

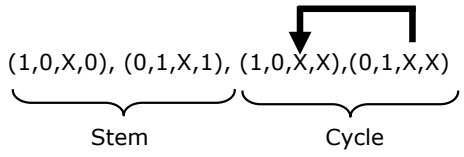

Fig. 2. Stem plus cycle example.

Consider the example in Figure 2. The system under analysis has four state bits, and the initial state is the vector $(1, 0, X, 0)$. After transitioning through the state $(0, 1, X, 1)$, the system will oscillate indefinitely between $(1, 0, X, X)$ and $(0, 1, X, X)$.

As the three valued simulation run is performed from the real initial state of the system, and the inputs applied for each state transition all are `X`s, we know that all the information on concrete values for state elements at particular time instances has to hold in every concrete run of the system. As a consequence, by inspecting the stem in Figure 2, we know that the first state bit in the second state of a run of the system must always be `0`. Furthermore, by looking at the stem and cycle, it is easy to see that the behavior of the first state bit can be generated by $[1, 0]^*$, and that the second state bit can be generated by $[0, 1]^*$. These state bits are thus clock-like.

Given a stem and cycle, we compute minperiod generators as follows:

1) `Candidates` = `Regs`, `RepeatLength` = 1
2) Remove all state elements from `Candidates` that assume the value `X` at some time point in the stem or cycle.
3) If `RepeatLength` does not divide the length of the cycle evenly, then `RepeatLength` = `RepeatLength` + 1, goto 3.
4) For each `Cand` $\in$ `Candidates`, check if the stem plus cycle for `Cand` shows that `Cand` can be generated by a `RepeatLength` generator. If so, assign `Cand` this generator, and remove `Cand` from further consideration.
5) `RepeatLength` = `RepeatLength` + 1. If the new `RepeatLength` is larger than the length of the cycle of the trace, terminate. Otherwise goto 3.

Note that our generator analysis may detect other looping system signals in addition to the clock signals. However, as all signal information we are detecting is valid data, our possible subsequent use of this free extra information can only increase the strength of our analysis. If clock annotations have been provided by the user, we add the corresponding generators to the computed set before proceeding to the next step of our analysis.

IV. Picking the number of phases

Given a set of extracted clock-like signal generators such as $G = \{R_0 = [0]^*, R_1 = [1, 0]^*, R_5 = [0, 1, 1]^*, R_7 = [1, 0, 1]^*\}$, our task is now to chose a number of phases `NumPhases` to abstract and then normalize the generators by (1) removing generators that can not be unrolled one or more times to form length `NumPhases` generators, and (2) expanding the remaining generators to length `NumPhases`.

In order to maximize the number of utilizable generators, we use the heuristic of choosing `NumPhases` to be the smallest number less than or equal to eight that maximizes the number of usable generators. We have found that this heuristic often allows us to keep a large fraction of the generators, while not forcing an excessive number of circuit unfoldings in the abstraction step. Analyzing G, we see that the optimal `NumPhases` is six, which leaves us with a normalized set of generators $G_{exp} = \{R_0 = [0, 0, 0, 0, 0, 0]^*, R_1 = [1, 0, 1, 0, 1, 0]^*, R_5 = [0, 1, 1, 0, 1, 1]^*, R_7 = [1, 0, 1, 1, 0, 1]^*\}$.

V. Construction of the abstracted design

By analyzing the original design D, we have extracted a heuristically picked number of phases `NumPhases` and a number of expanded generators G_{exp}. Our job is now to compute a reduced design.

Assume that D has a single output O and inputs $I_1 \ldots I_n$. We will create a new design D' that takes `NumPhases` time steps at a time. Each time step in this transformed design covers all of the different phases in the old design, so we

will need NumPhases outputs to track the status of O in each phase.

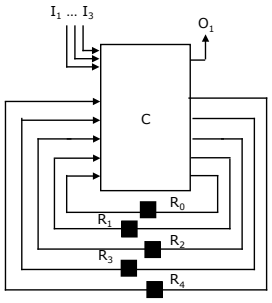

Fig. 3. Two phase rewriting example.

Consider the transformation of the design in Figure 3 for NumPhases $= 2$. The combinational logic C that computes the output O and next state values S_{i+1} from the inputs and current state values S_i will become a new block C'. This block differs from C in that it computes the register values S_{i+2} two steps into the future rather than one. Moreover, (1) the output O in the original design will become two new outputs O_0, and O_1 that track the value of O in phases zero and one, and (2) there will possibly be more than one instance of each original input.

We compute the new block as follows:

1) Cascade NumPhases copies of C.
2) For each generator for a state variable S_x, force the first value in the generator word on time instance 0 of S_x, the second on time instance 1 and so on.
3) Apply a fast logical minimizer to propagate the forced logical values and rewrite the cascaded logic block to as compact a representation as possible.
4) Trace back recursively from the output signals in the resulting C' to find the gates that are in the cone of influence. Remove everything else.

The transformed design D' can now be generated by reconnecting the surviving current states inputs to the next state outputs through state holding elements. The reset state for the new design is the projection of the original reset state to the registers left in the new design.

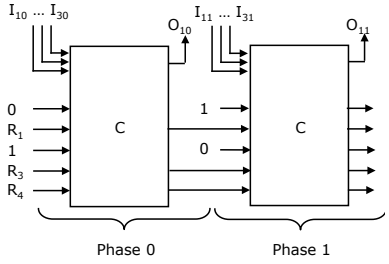

Fig. 4. Applying generator constants.

As an example, the two-phase abstraction of the five register circuit in Figure 3 will proceed as follow. Assume that our previous analysis steps have determined that $R_0 = [0,1]^*$ and that $R_2 = [1,0]^*$. After unwinding the design for two time steps, we apply the generator constants to the appropriate points as shown in Figure 4.

Next, the fast boolean minimizer is applied, and the cone of influence of the outputs is found. After having removed gates that do not occur in the cone of influence, we are left with the circuit in Figure 5. As can be seen, I_{10}, I_{20}, and I_{31} was

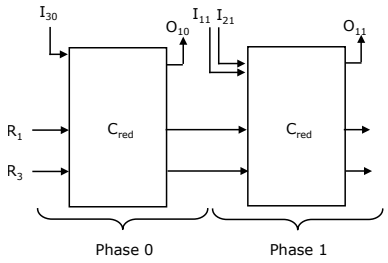

Fig. 5. Final result.

not in the cone of influence, nor was the current state input for R_4. As a consequence, the two-phase abstraction leaves us with a design with only two registers compared to the original five. Moreover, we have a total of three inputs: one phase zero input instance, and two phase one instances.

It is easy to relate the correctness of a phase abstracted design with the correctness of the original design:

Theorem 1: Assume that we have abstracted a design with N phases. The single original output O is then stuck at one precisely if $O_0 \ldots O_{N-1}$ are stuck at one.

Proof: (Sketch). The only transformation we have done is to unwind the circuit several times, to simplify away constants that we can safely assume, and to perform a cone of influence reduction. ■

Given that we find a length one or longer input stimuli sequence PhaseAbsCe that drives the phase k output to zero on a phase abstracted system with N phases, we can transform it to a stimuli sequence on the original system by performing the following steps:

1) RealCe $= [\,]$
2) InpStimuli = $head$(PhaseAbsCe)
3) If $length$(PhaseAbsCe) $= 1$, then LastPhase $= k$ else LastPhase $= N - 1$
4) For each phase p between 0 and LastPhase in order.
   a) Extract all input valuations belonging to phase p inputs from InpStimuli.
   b) Insert these input valuations into a new stimuli entry CurrStimuli
   c) Append CurrStimuli to RealCe
5) PhaseAbsCe = $tail$(PhaseAbsCe)
6) If PhaseAbsCe is empty, return RealCe, otherwise goto 2.

As an example, assume we have generated the length two counterexample

$$[[I_{30} = 1, I_{11} = 0, I_{21} = 0], [I_{30} = 0, I_{11} = 0, I_{21} = 0]]$$

for the D' corresponding to the abstraction in Figure 5. Furthermore, assume that this counterexample demonstrates that output O_{10} is not to stuck at one. The length three stimuli sequence

$$[[I_3 = 1], [I_1 = 0, I_2 = 0], [I_3 = 0]]$$

will then drive O_1 to the value zero.

VI. Theoretical results

The effectiveness of phase abstraction can be determined by observing the reduction in the number of state variables, and comparing that to the increase (if any) in the amount of combinational gates and inputs.

A categorical answer for how effective our analysis will be is not easy to give, as we are relying on a quick heuristic for identifying clock networks, and a fast but incomplete logic simplifier. The worst case for our analysis occurs when no clock-like signals are found or the post-unrolling simplification provide very little reduction. In this case, although the new system has at most as many state-holding elements as the original system, it will have about N times as many logical gates, and N times as many inputs and outputs. Our hope is that after abstraction, the reduction will have led to a system that has a comparable (or fewer) number of gates to the original system, a comparable number of inputs, and significantly fewer state-holding elements.

Let us look what the ideal results are. Assume that we are processing a strict N phase design, and that we perform perfect identification of clocking networks and optimal logic minimization. In this case all the state elements that are not active in phase 0 will be removed. The reason for this is that our new system is unfolded so that values are registered every Nth cycle. As the only state-holding elements that are nontransparent at time $0, N, 2N, 3N \ldots$ are the phase 0 elements, the remaining elements will not be in the cone of influence of the outputs. Moreover, as inputs are only read in the first phase of the design, the phase abstracted design will have at most as many inputs as the original design.

VII. Experimental results

When we introduced our implementation of the algorithms in Sections III, IV, and V as an extra model reduction step in our property checking tool, we found that the resulting abstraction was instrumental in solving a number of previously unsolvable problems at customer sites. In this section we will study the detailed results on some of these examples. Our implementation of the fast minimization pass uses Binary Expression Diagrams (BEDs) [4]. No setup, manual clock identification, or changes to our standard model checking flow has been done in our experiments. All the examples are from different sources, so no assumptions have been made about coding styles. In fact, before the algorithm was run, we had no idea of whether or not there were latches or flip-flops in the design, whether there were more than one internal clock used, or how internal clock signals (if any) were related.

Design	Phases	Size before (Regs/Inps/Gates)	Size after (Regs/Inps/Gates)
D_1	2	139/53/3.5k	80/53/3.3k
D_2	1	928/37/12k	667/37/8.9k
D_3	6	1063/49/8.3k	595/53/8.8k
D_4	2	1764/383/38k	1287/342/28k
D_5	2	20k/20k/847k	7k/7k/40k

TABLE I
Phase abstraction results

A. Size reduction

Before applying the phase abstraction pass, we compact the circuits as far as we can using all the model compression technology we have at our disposal. Our baseline representation is thus as small as we can get it.

As can be seen in Table I, the size of the original designs range from a little bit more than a hundred state-holding elements to more than twenty thousand registers. Our overall result is a 41% reduction in the number of registers on average. The longest abstraction time is a few minutes or so.

Consider the case of design D_1, which starts out with 139 registers, 53 inputs, and about 3500 gates. We detect two phase clocking, and after abstraction the resulting system contains only 80 registers. By performing our abstraction we have hence arrived at a system that does twice as much work in each time cycle, but that uses only 60% of the original registers. Our abstracted system has as many inputs as the original system, and contain slightly fewer gates.

Although Table I uniformly shows powerful reductions in the number of state variables, we are not guaranteed to decrease the gate count or number of inputs. In the cases of D_3, we see a slight increase both in the number of inputs and the number of gates. However, this is a very small price to pay for the significant decrease in the number of registers. That is not to say that we can not decrease the gate count significantly in some cases. In the case of D_5 the number of gates decrease by a factor of 20, and we remove two thirds of the registers.

It is interesting to observe that we can gain a lot by our analysis, even when we do not detect clock-like signals with periods greater than one. Consider the case of D_2. Here we get rid of 250 registers by sequentially propagating signals we detect are stuck at some constant in the three valued simulation. Note that extracting this information is nontrivial using standard methods—none of our standard compression tricks could detect these constants. This is not surprising, as we have to study the circuit behavior over seven time instances to prove that these signals are stuck.

B. Effect on later analyses

Previous work has already established that phase abstraction can increase the verification capacity of model checking by several orders of magnitude [2]. However, there has been very little analysis of where this increase in capacity comes from. In our flow, we see positive synergies with many different engines.

First of all, as previously discussed, localization reduction [5] followed by the use of BDDs to do fixpoints is highly sensitive to the number of registers in the abstraction. If we are removing close to 50% of the registers, it is clear that many properties that are out of reach may become tractable. As an example, D_4 can be solved in ten minutes using localization reduction after abstraction but does not terminate without it. Note, though, that the positive effect of phase abstraction on localization reduction need not only come from reducing the number of state bits that have to appear in abstractions—another dimension is the reduction in choices for refinement candidates. If a successful localization on a phase abstracted system with N phases requires k registers, a localization on the original design is likely to require on the order of $N * k$ correctly made choices. The abstraction has thus dramatically reduced the number of erroneous guesses possible.

Second, reducing the amount of redundancy in the design have positive effects on many analyses such as induction [6] and interpolation based model checking [7] that prove properties without localization. To capitalize on this, many approaches for making induction more powerful, such as van Eijk's algorithm [8] or the REVERSE approach to verification of retimed designs [9], strengthen the property under analysis with invariants about the design. Our analysis can be viewed as the detection of cyclical signal invariants which are incorporated directly into the model rather than as a property strengthening. Moreover, the detection of sequential constants in itself sometimes prove certain properties.

Third, even in the case of bughunting using Bounded Model Checking (BMC) [10], the use of phase abstraction is beneficial. This may not be obvious, as the number of state variables in a given system is not as much of a bottleneck for SAT as in the case of BDD-based analyses. However, consider the fact that in a successfully abstracted N phase system, a comparable amount of logic is used to cover multiple time steps in the old system. As a result, the same amount of search space can often be covered in much smaller time. As an example, in the case of D_3 only 362 cycles of bounded proof can be done overnight for the original system, compared to 432 cycles for the abstracted version.

Fourth, there are also good synergies possible with other transformations such as retiming [11]. Although design D_1 is very small, all the 139 registers are necessary for a proof so it is far from trivial to solve. Retiming interleaved with combinational rewriting (as described in [12]) before phase abstraction can not get rid of more than a few registers, due to loops in the design locking in registers. After abstraction, retiming can reduce the design to a 33 register machine that can be solved by BDD fixpoints in a few seconds.

VIII. Extensions

A prime difference between our algorithm in the previous sections, and the algorithms presented in [2] and [3], is that Baumgartner's method for processing an MDL can remove latches from any phase. In the presentation in Section V, we always keep the phase 0 latches, but we are in reality not restricted to this choice: In Section V we compute a system that transition from phase 0 in one clock sequence to phase 0 in the next clock sequence. By rotating the generators so that we compute a system that instead transitions from phase i to phase i for some $i \in \{1..N-1\}$, a different set of registers will be non-transparent at the snapshot time. Note, though, that if we choose $i \neq 0$, we have to perform a little bit of extra work. First of all, to preserve soundness we have to check the status of O in the first $i-1$ time instances for the original system separately. Second, as our system now starts in a phase $i > 0$, we have to compute a symbolic initial state that encodes all the possible values for these registers at time i in the original design.

Phase abstraction with a different base phase than zero has the potential to improve results significantly. However, in many cases the size of the result does not vary very much based on which phase we decide to keep—for D_3 the number of final state elements varies with less than 10 registers.

IX. Related work

As we have demonstrated, phase abstraction is a very powerful technique. However, it has not been studied extensibly in the literature, probably due to the previous reliance on strict design styles. The first work we are aware of is that of Hasteer *et al.* on the use of phase abstraction during sequential hardware equivalence checking [1]. Baumgartner and coworkers' papers extends Hasteer's work to the domain of general property checking and adds some additional refinements ([2], [3]). As outlined in Section I, the principal advantages of our approach over the previously proposed methods is that our algorithm

- does not rely on any knowledge of design styles
- is completely general in terms of the structure of designs that can be abstracted, and the permitted clocking styles
- does not rely on weak syntactic analyses
- provides an automatic way to identify clocks and their relationships but is not restricted to only use automatically generated information
- can detect non-clock signals who can be used to further simplify the design

In a later paper Baumgartner and coworkers considers the abstraction of what they call *C-slow* designs—designs which can be thought of as c copies of a basic design running on independent problems at the same time ([11], [13]). Formally, C-slow netlists are netlists comprised only of *simple* flip-flops (FFs) and gates, whose gates and state holding elements can be colored using some number c colors in such a way that

1) Each FF is assigned some color.
2) All FFs which have FFs of color i in their support have color $(i+1) \bmod c$.
3) All gates which have FFs of color i in their support have color i.

Here, simple FFs are flip flops clocked by a single master clock without clock gating. Note that the above definition allows inputs to be read by more than one type of FF. The result of

C-slow abstraction is a netlist that only maintains FFs of some particular color.

At a first glance, C-slow reduction might seem like a generalization of Baumgartner's strict dual phase abstraction algorithm to N phases with an added relaxation on how inputs can be read. However, as discussed in [13] it is not: C-slow reduction abstracts a design that is processing c independent problems *concurrently*, whereas phase abstraction transforms a netlist that is processing a single piece of data over c clock cycles. The two analyses are thus completely orthogonal. In fact, if a design contains multiple clocks or clock gating, phase abstraction is a necessary precondition to C-slow reduction. Moreover, note that our extensions to phase abstraction are necessary to allow C-slow reduction of designs with clock gating—Baumgartner's phase abstraction algorithm cannot deal with such circuits.

In [14], Albrecht and Hu presents a theory for reasoning about register transformations in the presence of multiple clock domains. Strict phase abstraction is one of a number of particular transformation rules that can be derived in this theory. Since our phase abstraction algorithm can not be stated as a simple register removal rule, our approach can not be viewed as a single similar derived rule. However, the individual rewrite steps that are performed during our analysis can be viewed as particular transformations in this theory.

X. Future work

Baumgartner's phase abstraction algorithm separates a design into MDLs, and each MDL is abstracted individually. Partly, this is by necessity as only regions that match a very specific design style can be processed. However, there are some advantages to partitioning in general. Notably, Baumgartner's algorithm gets to make the choice of removing ϕ_1 or ϕ_2 latches individually for each MDL. In contrast, our algorithm requires a global choice even if a design can be partitioned into multiple regions. While we have not seen much benefit from choosing other base phases than zero for abstraction, we are still interested in exploring ways to separate a design into non-MDL regions, abstract each region individually, and stitching together the results into a design whose size is smaller than the abstraction of the whole machine.

XI. Conclusions

In this paper, we have extended the previous state of the art for phase abstraction as represented by Baumgartner and coworkers work in [2] and [3] significantly. While the original treatise on phase abstraction for model checking showed that phase abstraction worked well in the context of a design style mandated within IBM, and multiphase clocking is used throughout the semiconductor industry, we are not aware of any mainstream adoption in formal verification tools. Our approach attempts to rectify this situation by extending Baumgartner and coworkers' work so that (1) the analysis becomes completely automatic, (2) we avoid the reliance on specific design styles, and (3) we bypass the syntactic approach to classifying the type of different state elements. Unlike the standard method, our algorithm can be integrated into any model checker in any flow. Moreover, our analysis is trivially correct as we are just performing a time transform of the system under analysis and simplifying away sequential constants.

Essentially our contribution is twofold: First of all, we have extended the set of designs amenable to phase abstraction significantly. Second, we have shown how to automate the analysis further by detecting clocks without user intervention. Note that while our clock identification strategy is not guaranteed to detect all clocks, our experimental results shows that our approach is powerful enough to detect a sufficient number of clocks to drastically reduce the complexity of testcases from a wide variety of customers. Moreover, if a user is unsatisfied with the automatic results and wants to manually identify clocks, we have not precluded this in any part of this presentation.

We hope that our work will make phase abstraction become a staple reduction for model checkers.

References

[1] G. Hasteer, A. Mathur, and P. Bannerjee, "A framework for equivalence checking of multi-phase FSMs," in *Proc. High Level Design Validation and Test Symp.*, 1997.

[2] J. Baumgartner, T. Heyman, V. Singhal, and A. Aziz, "Model checking the IBM gigahertz processor: An abstraction algorithm for high-performance netlists," in *Proc.* 11^{th} *Int. Conf. on Computer Aided Verification*, 1999.

[3] ——, "An abstraction algorithm for the verification of level-sensitive latch-based netlists," *Formal Methods in System Design*, vol. 23, pp. 39–65, 2003.

[4] H. R. Andersen and H. Huulgaard, "Boolean expression diagrams," in *Proc.* 12^{th} *IEEE Symp. on Logic in Computer Science*, 1997.

[5] R. Kurshan, *Computer Aided Verification of Coordinating Processes*. Princeton University Press, 1994.

[6] M. Sheeran, S. Singh, and G. Stalmarck, "Checking safety properties using induction and a SAT-solver," in *Proc. FMCAD '00,* 2^{th} *Int. Conf. on Formal Methods in Computer-Aided Design*. Lecture Notes in Computer Science, 2000.

[7] K. McMillan, "Interpolation and SAT-based model checking," in *Proc.* 15^{th} *Int. Conf. on Computer Aided Verification*. Lecture Notes in Computer Science, 2003.

[8] P. Bjesse and K. Claessen, "SAT-based verification without state space traversal," in *Proc. FMCAD '00,* 2^{th} *Int. Conf. on Formal Methods in Computer-Aided Design*. Lecture Notes in Computer Science, 2000.

[9] M. Mneimneh and K. Sakallah, "REVERSE: Efficient sequential verification for retiming," in *Proc. Int. Workshop on Logic Synthesis*, 2003.

[10] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, "Symbolic model checking using SAT procedures instead of BDDs," in *Proc.* 37^{th} *Design Automation Conference*, 1999.

[11] C. Leiserson and J. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, pp. 5–35, 1991.

[12] J. Baumgartner and A. Kuehlmann, "Min-area retiming on flexible circuit structures," in *Proc.* 13^{th} *Int. Conf. on Computer Aided Verification*. Lecture Notes in Computer Science, 2001.

[13] J. Baumgartner, A. Tripp, A. Aziz, V. Singhal, and F. Andersen, "An abstraction algorithm for the verification of generalized C-slow designs," in *Proc.* 12^{th} *Int. Conf. on Computer Aided Verification*. Lecture Notes in Computer Science, 2000.

[14] A. Albright and A. Hu, "Register transformations with multiple clock domains," in *Proc. Correct Hardware Design and Verification Methods, 11th IFIP WG 10.5 Advanced Research Working ConferenceProc.* Lecture Notes in Computer Science, 2000.