Exploiting Constraints in Transformation-Based Verification

Hari Mony[1,2], Jason Baumgartner[1], and Adnan Aziz[2]

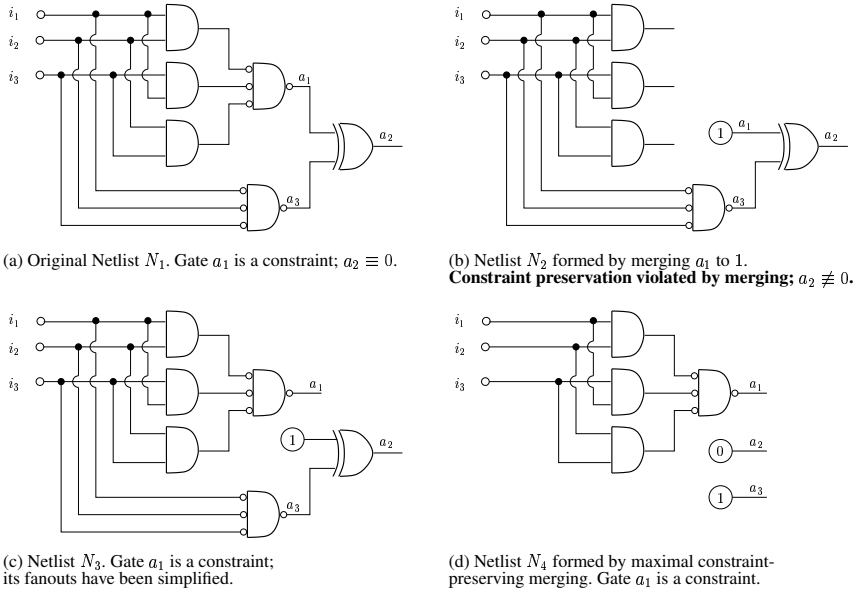
[1] IBM Systems & Technology Group [2] The University of Texas at Austin

Abstract. The modeling of design environments using constraints has gained widespread industrial application, and most verification languages include constructs for specifying constraints. It is therefore critical for verification tools to intelligently leverage constraints to enhance the overall verification process. However, little prior research has addressed the applicability of transformation algorithms to designs with constraints. Even when addressed, prior work lacks optimality and in cases violates constraint semantics. In this paper, we introduce the theory and practice of *transformation-based verification* in the presence of constraints. We discuss how various existing transformations, such as redundancy removal and retiming, may be optimally applied while preserving constraint semantics, including *dead-end states*. We additionally introduce novel constraint elimination, introduction, and simplification techniques that preserve property checking. We have implemented all of the techniques proposed in this paper, and have found their synergistic application to be critical to the automated solution of many complex verification problems with constraints.

1 Introduction

Constraints are pervasively used across a variety of verification frameworks. For example, the compositional verification framework advocates verifying a system by checking properties of its components using *assume-guarantee* reasoning. The assumptions that a component's environment needs to satisfy are often modeled using constraints. The modeling of verification environments using constraints has gained widespread industrial acceptance [1], and most industrial verification languages include constructs to specify constraints – for example, PSL [2], CBV [3], and *e* [4]. Constraints are also used to implement case-splitting strategies to enhance complex verification tasks, for example, arithmetic and datapath correctness [5, 6].

Given their pervasiveness, it is important for verification algorithms to leverage constraints to enhance the overall verification process. However, it is even more critical to preserve constraint semantics during this process. The concept of *transformation-based verification* (TBV) has been proposed to synergistically apply various automated transformation algorithms to simplify and decompose complex problems into simpler problems which may be solved with exponentially lesser resources [7, 8]. However, little prior research has addressed the applicability of various transformation algorithms in the presence of constraints. Additionally, in some cases prior research lacks optimality, and does not even guarantee the preservation of constraint semantics. For example, an approach for simplifying a combinational netlist in the presence of constraints is proposed in [9] as part of a Boolean-reasoning framework, which suffers these weaknesses.



(a) Original Netlist N_1. Gate a_1 is a constraint; $a_2 \equiv 0$.

(b) Netlist N_2 formed by merging a_1 to 1.
Constraint preservation violated by merging; $a_2 \not\equiv 0$**.**

(c) Netlist N_3. Gate a_1 is a constraint;
its fanouts have been simplified.

(d) Netlist N_4 formed by maximal constraint-preserving merging. Gate a_1 is a constraint.

Fig. 1: Combinational constraint example

Constraint-preserving testcase generation for simulation has been widely researched, e.g., in [10, 11]. These solutions, however, do not address preservation of *dead-end constraints* which entail states for which there is no legal input stimulus. Dead-end constraints tend to reduce the efficiency of explicit-state analysis, as well as semi-formal search; when a dead-end state is reached, the only recourse is to backtrack to an earlier state. Though dead-end constraints are considered *user errors* in certain methodologies [10], they are specifiable in a variety of languages, and in cases are powerful constructs for modeling verification tasks and case-splitting strategies [5].

Constraint Challenges to TBV. Constraints specify conditions that must hold in any state explored by a verification algorithm. To illustrate the impact of constraints, consider the combinational netlist illustrated in Figure 1. In the original netlist N_1, gate a_2 could evaluate to 1 (e.g., if $i_1 = 1$ and $i_2 = i_3 = 0$) or 0 (e.g., if $i_1 = i_2 = i_3 = 0$). However, labeling gate a_1 as a constraint would force at least two of i_1, i_2, i_3 to evaluate to 1, in turn forcing gate a_3 to evaluate to 1 and a_2 to evaluate to 0. For optimality, it is desirable to leverage the constraint to simplify the netlist accordingly. In [9], a structural conjunctive decomposition of the constraint is proposed, traversing each constraint gate fanin-wise through AND gates and stopping at inversion points and other gate types, merging each of these terminal gates to constant ONE. Applying this algorithm to netlist N_1, gate a_1 will be merged to constant ONE. However, this merging *fails to preserve constraint semantics* as gate a_2 in the resulting netlist N_2 could evaluate to 1 (if $i_1 = i_2 = i_3 = 0$). This demonstrates that redundancy removal applications must take precautions when leveraging constraints to increase their reduction potential.

In a sequential netlist, constraints pose additional challenges as illustrated by the example depicted in Figure 2. Constraint *c* disallows precisely the input sequences that can evaluate *t* to 1. If j $>$ i, then *t* can evaluate to 1 as the constraint precludes such

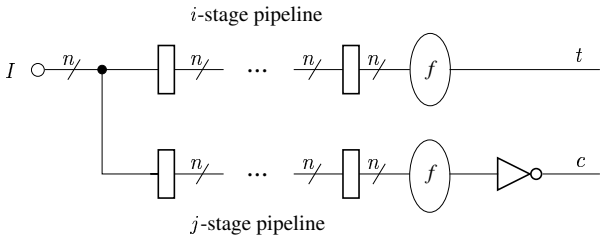


Fig. 2: Sequential constraint example

paths only at a later time-step. If on the other hand j $\leq$ i, constraint c prevents t from ever evaluating to 1. This demonstrates that temporal abstractions like retiming [7], which may effectively alter the values of i and j, must take precautions to ensure that constraint semantics are preserved through their transformations.

Contributions. In this paper we make several fundamental contributions to improving the efficiency of constraint-based verification frameworks [1].

1. We are the first to discuss how various existing automated transformation algorithms may be optimally applied in a property-preserving manner to designs with constraints. Table 1 enumerates these transformations, along with an overview of the corresponding challenges and solutions. *Overapproximation* refers to the risk of the transformation yielding spurious counterexamples. *Underapproximation* refers to the risk of the transformation yielding an incorrect proof of correctness.
2. We introduce fully-automated techniques for eliminating, introducing, and simplifying constraints in a property-preserving manner, enumerated in Table 2.

We have implemented all of these techniques in a verification toolset. We have found each of these techniques to be useful in the verification of designs with constraints. Furthermore, we have found that the *synergistic* application of these techniques is capable of yielding dramatic improvements to the verification of such designs, enabling conclusive results to problems that we have otherwise found unsolvable. Though we focus on the application of these techniques to formal verification, their structural nature enables their benefits to arbitrary frameworks, including testcase generation and synthesis.

Section	Technique	Challenge	Solution
3	Redundancy Removal	Merging within constraint cones may lead to overapproximation.	Disallow merging within a constraint cone, if redundancy proof *requires* that constraint.
4	Retiming	Varying lags of targets and constraints may lead to overapproximation as well as underapproximation.	Force identical lags across all target and constraint gates in retiming graph. Re-apply unfolded constraints to recurrence structure.
5	Target Enlargement	Transition-function based methods may lose correlation between constraint and target cones, leading to overapproximation.	Force application of constraints to each functional preimage prior to input quantification.
6	Reparameterization	Dead-end states may be lost through the transformation, leading to overapproximation.	Re-apply dead-end states as a simplified constraint.
7	Phase Abstraction	State folding may cause underapproximation if targets and constraints are of different *phase*.	Methodologically require all targets and constraints to be of the same phase.
8	C-Slow Abstraction	State folding may cause underapproximation if targets and constraints are of different *colors*.	Methodologically require all targets and constraints to be of the same color.
8	C-Slow Abstraction	Abstraction loses correlation across differing mod-c time-frames, causing overapproximation.	Methodologically disallow constraints that are not amenable to mod-c reasoning.

Table 1: Contributions to enable transformations in the presence of constraints

Section	Technique	Description
10	Constraint Elimination	Replace the constraint with an *accumulator circuit* to *remember* whether the constraint signal has been previously violated; conjunct to the target.
11	Constraint Introduction	Attempt to derive conditions after which targets are never hittable; add as constraints.
12	Constraint Simplification	Attempt to replace a constraint with its preimage, to reduce the size of its cone and enable its elimination through reparameterization.

Table 2: Constraint transformation contributions

2 Formalisms

In this section, we provide formalisms used throughout the paper. A reader well-versed in hardware verification may wish to skip this section, using it as a reference.

Definition 1. A *netlist* is a tuple $N = \langle\langle V, E\rangle, G, T, C, Z\rangle$ comprising a finite directed graph with vertices V and edges $E \subseteq V \times V$, a semantic mapping from vertices to gate types $G : V \mapsto types$, a set of *targets* $T \subseteq V$ correlating to a set of properties $AG(\neg t), \forall t \in T$, and a set of *constraints* $C \subseteq V$. The function $Z : V \mapsto V$ is the symbolic initial value mapping.

Our verification problem is represented entirely as a netlist, comprising the *design under verification*, its *environment*, and its *property automata*. Our gate $types$ define a set of primary inputs, registers (our only sequential gate type), and combinational gates with various functions, including constants. The type of a gate may place constraints upon its incoming edge count – e.g., each register has an indegree of one (whose source gate is referred to as its *next-state function*); primary inputs and constants have an indegree of zero. We denote the set of inputs as $I \subseteq V$, and the set of registers as $R \subset V$. The initial values of a netlist represent the values that registers can take at time 0. We disallow registers from appearing in any initial value functions. Furthermore, we do not allow combinational cycles in a legal netlist.

Definition 2. The *semantics of a netlist* N are defined in terms of semantic traces. 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 functions from $V \times \mathbb{N}$ to $\{0, 1\}$ which are consistent with the following rule. The value of gate v at time i in trace p is denoted by $p(v, i)$. Term u_j denotes the source vertex of the j-th incoming edge to v, implying that $(u_j, v) \in E$.

$$p(v,i) = \begin{cases} s^i_{v_p} & : v \text{ is a primary input with sampled value } s^i_{v_p} \\ G_v\big(p(u_1,i),...,p(u_n,i)\big) & : v \text{ is a combinational gate with function } G_v \\ p(u_1, i-1) & : v \text{ is a register and } i > 0 \\ p\big(Z(v), 0\big) & : v \text{ is a register and } i = 0 \end{cases}$$

The length of a trace p is defined as $length(p) = \min\{i : \exists c \in C.\ p(c, i) = 0\}$. A target t is said to be hit in a trace t at time i iff $\big(p(t, i) = 1\big) \wedge \big(i < length(p)\big)$. We define $hit(p, t)$ as the minimum i at which t is hit in trace p, or -1 if no such i exists.

Definition 3. Netlists N and N' are said to be *property-preserving trace equivalent* with respect to target sets T and T' respectively, iff there exists a bijective mapping $\psi : T \mapsto T'$ such that:

- $\forall p \in P. \exists p' \in P'. \forall t \in T. \big(hit(p, t) = hit(p', \psi(t))\big)$
- $\forall p' \in P'. \exists p \in P. \forall t \in T. \big(hit(p, t) = hit(p', \psi(t))\big)$

1. Guess the *redundancy candidates* – i.e., suspected-equivalent gate sets.
2. Attempt to prove that each pair of candidates is truly equivalent.
3. If any of the candidate pairs cannot be proven equivalent, refine them and goto Step 2.
4. The redundancy candidates are accurate; the corresponding gates may be merged.

Fig. 3: Generic redundancy removal algorithm

3 Redundancy Removal

Redundancy removal (e.g., [9, 12]) is the process of demonstrating that two gates in a netlist always evaluate to the same value. Once a pair of redundant gates are identified, the netlist may be simplified by *merging* one of the gates onto the other; i.e., by replacing each fanout reference to one gate by a reference to the other. For property checking, it is sufficient to reason about the prefix *length* of a trace as per Definition 2. Constraints therefore generally cause more gates to appear redundant (within this prefix) than otherwise. For optimality, redundancy removal algorithms should thus leverage the constraints to increase their reduction potential. For example, when using the framework of Figure 3, the algorithms which identify redundancy candidates in Step 1 and the algorithms which prove each of the candidates redundant in Step 2 must leverage the constraints to avoid a loss of reduction potential. However, as per Figure 1b, once redundant gates have been identified, proper care must be taken while merging them to avoid violating constraint semantics.

Theorem 1. Consider gate u which is not in the cone of constraint set $U \subseteq C$. Gate u may be merged onto any other gate v while preserving property checking provided that the proof of $u \equiv v$ does not require the trace-prefixing effect of constraints $C \setminus U$.

Proof. (Sketch) Since $u \equiv v$ within all valid trace prefixes, the only risk of violating property checking due to this merge is that the constraining power of a constraint gate is diminished as per Figures 1a-1b. By Definition 2, the merge of u onto v only alters the evaluation of gates in the fanout of u. However, since the trace-prefixing effect of no constraint in the fanout of u was leveraged to enable the merge, this merge cannot diminish the constraining power of the resulting netlist. □

Theorem 1 illustrates that gates outside of the cone of the constraints may be merged without violating constraint semantics, though care must be taken when merging gates within the cones of the constraints to ensure that their constraining power is not diminished. Netlist N_4 of Figure 1d illustrates the result of optimal property-preserving redundancy removal of netlist N_1. In Section 6, we will address the property-preserving elimination of gates within the cones of constraints whose trace-prefixing may be used to enable that elimination via the technique of structural reparameterization.

4 Retiming

Retiming is a synthesis optimization technique capable of reducing the number of registers of a netlist by relocating them across combinational gates [13].

Definition 4. A *retiming* of netlist N is a gate labeling $r : V \mapsto \mathbb{Z}$, where $r(v)$ is the *lag* of gate v, denoting the number of registers that are moved backward through v. A *normalized retiming* r' may be obtained from an arbitrary retiming r, and is defined as $r' = r - \max_{v \in V} r(v)$.

In [7], normalized retiming is proposed for enhanced invariant checking. The retimed netlist $\tilde{N}$ has two components: **(1)** a sequential *recurrence structure* $\tilde{N}'$ which has a unique representative for each combinational gate in the original netlist N, and whose registers are placed according to Definition 4, and **(2)** a combinational *retiming stump* $\tilde{N}''$ obtained through unfolding, representing retimed initial values as well as the functions of combinational gates for prefix time-steps that were effectively discarded from the recurrence structure. It is demonstrated in [7] that each gate $\tilde{u}'$ within $\tilde{N}'$ is trace-equivalent to the corresponding u within N, modulo a temporal skew of $-r(u)$ time-steps. Furthermore, there will be $-r(u)$ correspondents to this u within $\tilde{N}''$, each being trace-equivalent to u for one time-step during this temporal skew. Property checking of target t is thus performed in two stages: a bounded check of the time-frames of t occurring within the unfolded retiming stump, and a fixed-point check of $\tilde{t}'$ in the recurrence structure. If a trace is obtained over $\tilde{N}'$, it may be mapped to a corresponding trace in N by reversing the $\langle$*gate, time*$\rangle$ relation inherent in the retiming.

Theorem 2. Consider a normalized retiming where every target and constraint gate is lagged by the same value $-i$. Property checking will be preserved provided that:

1. the i-step bounded analysis of the retiming stump enforces all constraints across all time-frames, and
2. every retimed constraint gate, as well as every unfolded time-frame of a constraint referenced in a retimed initial value in $\tilde{N}'$, is treated as a constraint when verifying the recurrence structure.

Proof. (Sketch) Correctness of (1) follows by construction of the bounded analysis. Correctness of (2) follows from the observation that: **(a)** every gate lagged by $-i$ time-steps (including all targets and constraints) is trace-equivalent to the corresponding original gate modulo a skew of i time-steps, and **(b)** the trace pruning caused by constraint violations within the retiming stump is propagated into the recurrence structure by re-application of the unfolded constraint gates referenced in the retimed initial values. □

The min-area retiming problem may be cast as a minimum-cost flow problem [13]. One may efficiently model the restriction of Theorem 2 by *renaming* the target and constraint gates to a single vertex in the retiming graph, which inherits all fanin and fanout edges of the original gates. This modeling forces the retiming algorithm to yield an optimal solution under the equivalent-lag restriction. While this restriction may clearly impact the optimality of the solution, it is generally necessary for property preservation.

5 Structural Target Enlargement

Target enlargement [14] is a technique to render a target t' which may be hit at a shallower depth from the initial states of a netlist, and with a higher probability, than the original target t. Target enlargement uses preimage computation to calculate the set of states which may hit target t within k time-steps. A transition-function vs. a transition-relation based preimage approach may be used for greater scalability. Inductive simplification may be performed upon the k-th preimage to eliminate states which hit t in fewer than k time-steps. The resulting set of states may be synthesized as the enlarged target t'. If t' is unreachable, then t must also be unreachable. If t' is hit in trace p', a corresponding trace p hitting t may be obtained by casting a k-step bounded search

Compute $f(t)$ as the function of the target t to be enlarged;
Compute $f(c_i)$ as the function of each constraint c_i;
$B_0 = \exists I.\big(f(t) \wedge \bigwedge_{c_i \in C} f(c_i)\big)$;
for ($k = 1$; $\neg\, done$; k++) { // Enlarge up to arbitrary termination criteria $done$
  If t may be hit at time $k-1$ while adhering to constraints, return the corresponding trace;
  $B_k = \exists I.\big(preimage(B_{k-1}) \wedge \bigwedge_{c_i \in C} f(c_i)\big)$;
  Simplify B_k by applying $B_0, \ldots, B_{k-1}$ as *don't cares*;
}
Synthesize B_k using a standard multiplexor-based synthesis as the enlarged target t';
If t' is proven unreachable, report t as unreachable;
If trace p' is obtained hitting t' at time j {
  Cast a k-step constraint-satisfying unfolding from the state in p' at time j to hit t;
  Concatenate the resulting trace p'' onto p' to form trace p hitting t at time $k + j$; return p; }

Fig. 4: Target enlargement algorithm

from the state hitting t' in p' which is satisfiable by construction, and concatenating the result onto p' to form p. The modification of traditional target enlargement necessary in the presence of constraints is depicted in Figure 4.

Theorem 3. The target enlargement algorithm of Figure 4 preserves property checking.

Proof. (Sketch) The constraint-preserving bounded analysis used during the target enlargement process will generate a valid trace, or guarantee that the target cannot be hit at times $0, \ldots, k-1$, by construction. To ensure that the set of enlarged target states may reach the original target along a trace which does not violate constraints, the constraint functions are conjuncted onto each preimage prior to input quantification. The correctness of *target unreachable* results, as well as the trace lifting process, relies upon the fact that there exists an k-step extension of any trace hitting t' which hits t as established in [14], here extended to support constraints. □

There is a noteworthy relation between retiming a target t by $-k$ and performing a k-step target enlargement of t; namely, both approaches yield an abstracted target which may be hit k time-steps shallower than the corresponding original target. Recall that with retiming, we retimed the constraints in lock-step with the targets. With target enlargement, however, we retain the constraints intact. There is one fundamental reason for this distinction: target enlargement yields sets of states which only preserve the *hittability* of targets, whereas retiming more tightly preserves trace equivalence modulo a time skew. This relative weakness of property preservation with target enlargement is due to its input quantification and preimage accumulation via the don't cares. If preimages were performed to *enlarge* the constraints, there is a general risk that a trace hitting the enlarged target while preserving the enlarged constraints may not be extendable to a trace hitting the original target, due to possible conflicts among the input valuations between the constraint and target cones in the original netlist. For example, a constraint could evaluate to 0 whenever an input i_1 evaluates to 1, and a target could be hittable only several time-steps after i_1 evaluates to 1. If we enlarged the constraint and target by one time-step, we would lose the unreachability of the target under the constraint because we would quantify away the effect of i_1 upon the constraint.

6 Structural Reparameterization

Definition 5. A *cut of a netlist* is a partition of V into two sets: $\mathcal{C}$ and $\overline{\mathcal{C}} = V \setminus \mathcal{C}$. A cut induces a set of *cut gates* $V_{\mathcal{C}} = \{u \subseteq \mathcal{C} : \exists v \in \overline{\mathcal{C}}.\big(((u,v) \in E) \vee (v \in R \wedge u = Z(v))\big)\}$.

Reparameterization techniques, e.g., [15], operate by identifying a cut of a netlist graph $V_{\mathcal{C}}$, enumerating the valuations sensitizable to that cut (its *range*), then synthesizing the range relation and replacing the fanin-side of the cut by this new logic. In order to guarantee soundness and completeness for property checking, one must generally guarantee that target and constraint gates lie on the cut or its fanout. Given parametric variables p^i for each cut gate $V^i_{\mathcal{C}}$, the range is computable as $\exists I.\bigwedge_{i=1}^{|V_{\mathcal{C}}|}\big(p^i \equiv f(V^i_{\mathcal{C}})\big)$. If any cut gate is a constraint, its parametric variable may be forced to evaluate to 1 in the range to ensure that the synthesized replacement logic inherently reflects the constrained input behavior. This cut gate will then become a constant ONE in the abstracted netlist, effectively being discarded.

While adequate for combinationally-driven constraints and a subset of sequentially-driven constraints, this straight-forward approach does not address the preservation of dead-end states. A postprocessing approach is thus necessary to identify those abstracted constraints which have dead-end states, and to re-apply the dead-end states as constraints in the abstracted netlist. This check consists of computing $\exists I.f(c_i)$ for every constraint gate c_i used to constrain the range. If not a tautology, the result represents dead-end states for which no input valuations are possible, hence a straight-forward multiplexor-based synthesis of the result may be used to create a logic cone to be tagged as a constraint in the abstracted netlist.

Theorem 4. Structural reparameterization preserves property checking, provided that any constraints used to restrict the computed range are re-applied as simplified dead-end constraints in the abstracted netlist.

Proof. (Sketch) The correctness of reparameterization without dead-end constraints follows from prior work, e.g., [15]. Note that reparameterization may replace any constraints by constant ONE in the abstracted netlist. Without the re-application of the dead-end states as a constraint, the abstracted netlist will thus be prone to allowing target hits beyond the dead-end states. The re-application of the dead-end states as a constraint closes this semantic gap, preserving falsification as well as proofs. □

To illustrate the importance of re-applying dead-end constraints during reparameterization, consider a constraint of the form $i_1 \wedge r_1$ for input i_1 and register r_1. If this constraint is used to restrict the range of a cut, its replacement gate will become a constant ONE hence the constraint will be effectively discarded in the abstracted netlist. The desired byproduct of this restriction is that i_1 will be forced to evaluate to 1 in the function of all cut gates. However, the undesired byproduct is that the abstracted netlist will no longer disallow r_1 from evaluating to 0 without the reapplication of the dead-end constraint $\exists i_1.(i_1 \wedge r_1)$ or simply r_1. Because this re-application will ensure accurate trace-prefixing in the abstracted netlist, the range may be simplified by applying the dead-end state set as *don't cares* prior to its synthesis as noted in [11].

7 Phase Abstraction

Phase abstraction [16] is a technique for transforming a *latch-based* netlist to a register-based one. A latch is a gate with two inputs (*data* and *clock*), which acts as a buffer when its *clock* is active and holds its last-sampled *data* value (or initial value) otherwise. Topologically, a k-phase netlist may be k-colored such that latches of color i may only combinationally fan out to latches of color $\big((i+1) \bmod k\big)$; a combinational gate acquires the color of the latches in its combinational fanin. A modulo-k counter is used to clock the latches of color $(j \bmod k)$ at time j. As such, the initial values of only the $(k-1)$ colored latches propagate into other latches. Phase abstraction converts one color of latches into registers, and the others into buffers, thereby reducing state element count and temporally *folding* traces modulo-k, which otherwise stutter.

Phase abstraction may not preserve property checking for netlists with constraints as illustrated by the following example. Assume that we have a 2-phase netlist with a target gate of color 1, and a constraint gate of color 0 which is unconditionally violated one time-step after the target evaluates to 1. Without phase abstraction, the target may be hittable since the constraint prefixes the trace only on the time-step after the target evaluates to 1. However, if we eliminate the color-0 latches via phase abstraction, the constraint becomes violated concurrently with the target's evaluation to 1, hence the target becomes unhittable. Nonetheless, there are certain conditions under which phase abstraction preserves property checking as per the following theorem.

Theorem 5. If each constraint and target gate is of the same color, phase abstraction preserves property checking.

Proof. (Sketch) The correctness of phase abstraction without constraints has been established in prior work, e.g., [16]. Because every constraint and target gate are of the same color i, they update concurrently at times j for which $\big((j \bmod k) = i\big)$. Phase abstraction will merely eliminate the stuttering at intermediate time-steps, but not temporally skew the updating of the constraints relative to the targets. Therefore, the trace prefixing of the constraints remains property-preserving under phase abstraction. □

Automatic approaches of attempting to establish the criteria of Theorem 5, e.g., via *padding* pipelined latch stages to the constraints to align them with the color of the targets, are not guaranteed to preserve property checking. The problem is that such approaches unconditionally delay the trace prefixing of the constraints, hence even a contradictory constraint which can never be satisfied at time zero – which thus renders all targets unhittable – may become contradictory only at some future time-step in the range $1, \ldots, (k-2)$. After phase abstraction, this delay will be either zero or one time-step; in the latter case, we have opened a hole during which phase abstracted targets may be hit, even if they are truly unhittable in the original netlist. Nonetheless, in most practical cases, one may methodologically specify their desired verification problem in a way that adheres to the criteria of Theorem 5.

8 C-Slow Abstraction

C-slow abstraction [17] is a state folding technique which is related to phase abstraction, though is directly applicable to register-based netlists. A c-slow netlist [13] has

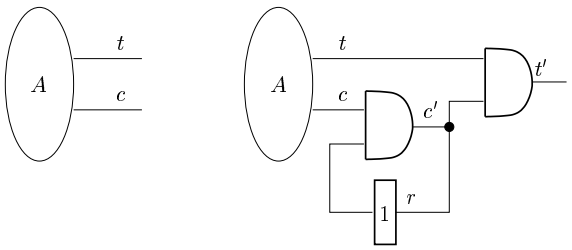
registers which may be c-colored such that registers of color i may only combinationally fan out to registers of color $\big((i+1) \bmod c\big)$; a combinational gate acquires the color of the registers in its combinational fanin. Unlike k-phase netlists, the registers in a c-slow netlist update every time-step hence generally never stutter. Additionally, the initial value of every register may propagate to other registers. C-slow abstraction operates by transforming all but a single color of registers into buffers, thereby reducing register count and temporally *folding* traces modulo-c. To account for the initial values which would otherwise be lost by this transformation, an unfolding approach is used to inject the retained registers into all states reachable in time-frames $0, \ldots, (c-1)$.

As with phase abstraction, if the target and constraint gates are of differing colors, this abstraction risks converting some hittable targets to unhittable due to its temporal collapsing of register stages. Additionally, even the criteria of requiring all target and constraint gates to be of the same color as with Theorem 5 is not guaranteed to preserve property checking with c-slow abstraction. The problem is due to the fact that c-slow netlists do not stutter $\bmod c$. Instead, each time-step of the abstracted netlist correlates to c time-steps of the original netlist, with time-steps $i, c+i, 2 \cdot c+i, \ldots$ being evaluated for each $i < c$ *in parallel* due to the initial value accumulation. Reasoning across $\bmod c$ time-frames is intrinsically impossible with c-slow abstraction; thus, in the abstracted netlist, there is generally no way to detect if a constraint was effectively violated at time $a \cdot c + i$ in the original netlist when evaluating a target at time $(a+1) \cdot c + j$ for $i \neq j$. Even with an equivalent-color restriction, c-slow abstraction thus risks becoming *overapproximate* in the presence of constraints. Nonetheless, methodologically, constraints which are not amenable to this state-folding process are of little practical utility in c-slow netlists. Therefore, in most cases one may readily map an abstracted counterexample trace to one consistent with the original netlist, e.g., using satisfiability analysis to ensure constraint preservation during intermediate timesteps.

9 Approximating Transformations

Overapproximating Transformations. Various techniques have been developed for attempting to reduce the size of a netlist by overapproximating its behavior. Any target proven unreachable after overapproximation is guaranteed to be unreachable before overapproximation. However, if a target is hit in the overapproximated netlist, this may not imply that the corresponding target is hittable in the original netlist. Localization [18, 19] is a common overapproximation technique which replaces a set of cut gates of the netlist by primary inputs. The abstracted cut can obviously simulate the behavior of the original cut, though the converse may not be possible.

Overapproximating transformations are directly applicable in the presence of constraints. Overapproximating a constraint cone only weakens its constraining power. For example, while the cone of target t and constraint c may overlap, after localizing the constraint cone it may only comprise localized inputs which do not appear within the target cone, thereby losing all of its constraining power on the target. Such constraint weakening is merely a form of overapproximation, which must already be addressed by the overall overapproximate framework. Both counterexample-based [18] and proof-based [19] localization schemes are applicable to netlists with constraints, as they will both attempt to yield a minimally-sized localized netlist such that the retained portion of the constraint and target cones will guarantee unreachability of the targets.



a) Netlist N; gate c is a constraint, gate t is a target.

b) Netlist N' with no constraints; gate t' is a target.

Eliminate_constraints (Netlist N)
$c = 1 \wedge \bigwedge_{v_i \in C} v_i$;
r = **create_register**;
$Z(r) = 1$;
$c' = c \wedge r$;
next_state(r) = c';
foreach $t \in T$
  $t' = t \wedge c'$;
replace each $t \in T$ by corresponding t';
remove all constraints: $C = \emptyset$;

c) Constraint elimination algorithm

Fig. 5: Property-preserving constraint elimination

Underapproximating Transformations. Various techniques have been developed to reduce the size of a netlist while underapproximating its behavior. For example, unfolding only preserves a time-bounded slice of the netlist's behavior; case splitting (e.g., by merging inputs to constants) may restrict the set of traces of a netlist. Underapproximating transformations may safely be applied to a netlist with constraints, as underapproximating a constraint cone only strengthens its power. For example, if a constraint is of the form $i_1 \vee i_2$, underapproximating by merging i_1 to constant ZERO will force i_2 to constant ONE in the underapproximated netlist even though a target may be hit in the original netlist only while assigning i_2 to a 0. However, this restriction – which may cause unreachable results for targets which were hittable without the underapproximation – must already be addressed by the overall underapproximate framework. Target hits on the underapproximated netlist still imply valid hits on the original netlist even in the presence of constraints. Extensions to underapproximate frameworks to enable completeness – e.g., diameter bounding approaches for *complete* unfolding, and complete case splitting strategies – are directly applicable in the presence of constraints.

10 Constraint Elimination

Given the challenges that they pose to various algorithms, one may wish to eliminate constraints in a property-preserving manner. In Figure 5c, we introduce a general constraint elimination algorithm.

Theorem 6. The constraint elimination algorithm of Figure 5c is a *property-preserving* transformation.

Proof. Consider any trace p that hits target t in netlist N at time i. Note that netlist N' has the same set of gates as N in addition to gates c', r, and t'. Consider the trace p' of N' where all common gates have the same valuations over time as in p, and gates c', r and t' are evaluated as per Definition 2. Because t is hit at time i, $\forall j \leq i.\big(p(c,j) = 1\big)$, and thus by construction of c', $\forall j \leq i.\big(p'(c',j) = 1\big)$. Because $t' = t \wedge c'$, we also have $\forall j \leq i.\big(p(t,j) = p'(t',j)\big)$. It may similarly be proven that for any trace p' that hits target t' at time i, there exists an equivalent trace p that hits target t at time i. □

Performing the constraint elimination transformation in Figure 5 enables arbitrary verification and transformation algorithms to be applied to the resulting netlist without

risking the violation of constraint semantics. However, this approach could result in significant performance degradation for both types of algorithms:

- Transformation algorithms (particularly redundancy removal) lose their ability to leverage the constraints for optimal simplification of the netlist.
- Falsification algorithms may waste resources analyzing uninteresting states, i.e., from which no target may subsequently be hit due to c' evaluating to 0.
- The tactical utility of the constraints for case-splitting strategies is lost.

11 Constraint Introduction

It follows from the discussion of redundancy removal in Section 3 that reduction potential may be increased by constraints. It may therefore be desirable to derive constraints that may be introduced into the netlist while preserving property checking, at least temporarily to enhance a particular algorithm.

Theorem 7. Consider netlist N with gate g. If no target in T may be hit along any trace after gate g evaluates to 0, then g may be labeled as a constraint while preserving property checking.

Proof. If gate g is labeled as a constraint, by Definition 2, we will only reason about the prefix *length* of traces wherein gate g always evaluates to 1. Since no target in T may be hit along any trace after gate g evaluates to 0, by Definition 3, netlist N' formed from N by labeling gate g as a constraint is property-preserving trace equivalent to N. □

Taking the example netlist of Figure 5b, any of the gates c, c', and r may be labeled as a constraint provided that we may establish the corresponding condition of Theorem 7, effectively reversing the transformation of Figure 5c. While this proof may in cases be as difficult as property checking itself, we propose an efficient heuristic algorithm for deriving such constraint candidate gates as follows. Similar to the approach of [20], we may localize each of the targets, and use a preimage fixed-point computation to underapproximate the number of time-steps needed to hit that target from a given set of states. Any state not reached during this fixed-point may never reach that target. The intersection of such state sets across all targets represents the conditions from which no target may subsequently be hit. While the approach of [20] proposes only to use this set to steer semi-formal analysis away from useless states, we propose to synthesize the resulting conditions as a constraint in the netlist to enhance reduction potential.

Note that these constraints are in a sense *redundant* because no target hits may occur after they evaluate to 0 anyway. Therefore, instead of forcing all algorithms to adhere to these constraints which may have an associated overhead, we may treat these as *verification don't cares* so that algorithms may choose to either use these constraints to restrict evaluation of the netlist, or to ignore them. Note that certain verification algorithms, e.g., SAT-based search, may inherently *learn* such conditions and direct their resources accordingly. Ours is a more general paradigm which enables leveraging this information for arbitrary algorithms, particularly to enhance reduction potential.

12 Constraint Simplification

In this section, we discuss a general approach to simplify constraints. We also discuss an efficient implementation of this paradigm which attempts to replace a constraint with

```
while (¬ done)   // Iterate until arbitrary termination criteria done
  Apply structural reparameterization to simplify constraint c;
  If constraint c has been eliminated by reparameterization, break;
  // Else, note that c has been simplified to its dead-end states
  If (prop_p(t, c) ≡ prop_p(t, struct_pre(c)))
    c = struct_pre(c);
  else break; // constraint c cannot be safely replaced by its preimage
```

Fig. 6: Heuristic constraint simplification algorithm

its preimage, heuristically trying to reduce the size of the constraint cone and enable the elimination of that constraint through reparameterization.

We define $prop\_p(t, c)$ as the target gate resulting from applying the constraint elimination algorithm of Figure 5c specifically to target t and gate c.

Theorem 8. Consider a netlist N with constraint c_1 and gate c_2. If $\forall t \in T.\big(prop\_p(t, c_1) \equiv prop\_p(t, c_2)\big)$ without the trace-prefixing of constraint c_1, then converting N into N' by labeling c_2 as a constraint instead of c_1 is a property-preserving transformation.

Proof. Since $\forall t \in T.\big(prop\_p(t, c_1) \equiv prop\_p(t, c_2)\big)$ without the trace-prefixing entailed by constraint c_1, this proof follows directly from Definition 3 and Theorem 6. □

Theorem 8 illustrates that in certain cases, we may modify the constraint gates in a netlist while preserving property checking. Practically, we wish to exploit this theorem to shrink the size of the constraint cones and thereby effectively strengthen their reduction potential. Note that the structural reparameterization algorithm in Section 6 is able to eliminate constraints which have no dead-end states. This is in a sense an optimal transformation, as the constraining power of the constraints are thereafter reflected in the netlist structure itself and effectively filters the input stimulus applied to the netlist. Given these motivations, we present a heuristic constraint simplification algorithm.

Definition 6. The *structural preimage* of a gate u which has no inputs in its combinational fanin, $struct\_pre(u)$, is a logic cone obtained by replacing each register gate $v \in R$ in the combinational fanin of gate u with its corresponding next-state function.

The algorithm of Figure 6 attempts to iteratively simplify, and ultimately eliminate, the constraints in a property-preserving manner. At each iteration, reparameterization is used to replace the current constraint by its dead-end states. Note that this step will eliminate the constraint if it entails no dead-end states. Otherwise, we attempt to simplify the resulting sequential constraint by replacing it with its structural preimage, using Theorem 8 to validate that this replacement preserves property checking. If this check fails (either through refutation or excessive resource requirements), then the algorithm terminates. Otherwise, the algorithm iterates with the resulting simplified constraint.

To illustrate how this algorithm works in practice, consider its application on constraint c in the netlist of Figure 2. If $j \leq i$, constraint c can be iteratively replaced by its preimage until it becomes combinational, at which point reparameterization will outright eliminate it. If $j > i$, constraint c can be simplified by shrinking j to $i + 1$, at which point the check based upon Theorem 8 fails causing the iterations to terminate.

Practically, the equality check of Figure 6 tends to be computationally expensive. However, this check can be simplified as per the following theorem.

Definition 7. The *structural initialization* of a gate u which has no inputs in its combinational fanin, $struct\_init(u)$, is a logic cone obtained by replacing each register gate $v \in R$ in the combinational fanin of gate u with its corresponding initial value function. The initial value constraint of u is defined as $init\_cons(u) = init\_r \vee struct\_init(u)$, where $init\_r$ is a register whose initial value is ZERO and next-state function is ONE.

Theorem 9. Consider a netlist N with constraint c_1. If $\forall t \in T.\big(prop\_p(t, c_1) \implies prop\_p(t, struct\_pre(c_1))\big)$ in N with the trace-prefixing entailed by constraint c_1, then converting N into N' by labeling $struct\_pre(c_1)$ and $init\_cons(c_1)$ as constraints instead of c_1 is a property-preserving transformation.

Proof. **(1)** The implication proof in N means that within the prefix of any trace, either the two gates evaluate to the same value, or $prop\_p(t, c_1)$ evaluates to 0 and $prop\_p\big(t, struct\_pre(c_1)\big)$ to 1. The latter condition cannot happen since within any prefix, constraint c_1 must evaluate to 1, which implies that t cannot evaluate to 1 and $prop\_p(t, c_1)$ to 0 concurrently. The implication proof thus ensures that if t is asserted within any prefix at time i, then $struct\_pre(c_1)$ must evaluate to 1 at times 0 to i.

(2) Since N and N' have the same set of gates, they also have the same set of traces; only the constraint sets differ. The trace prefixing of N' is stricter than that of N as follows. **(a)** All traces prefixed at time 0 because of constraint c_1 in netlist N are also prefixed at time 0 because of constraint $init\_cons(c_1)$ in N'. **(b)** All traces prefixed at time $i+1$ because of constraint c_1 in netlist N are prefixed at time i because of constraint $struct\_pre(c_1)$ in N'.

(3) For property-preservation, we must only ensure that target t cannot be asserted during time-steps that were prefixed in N' but not N. During such time-steps, c_1 evaluates to 1, and $struct\_pre(c_1)$ to 0, hence $prop\_p\big(t, struct\_pre(c_1)\big)$ must evaluate to 0. The proof of this implication check thus requires $prop\_p(t, c_1)$ to evaluate to 0 at such time-steps, ensuring that t evaluates to 0. □

Practically, we have found that the trace-prefixing of c_1 substantially reduces the complexity of the proof obligation of Theorem 9 vs. Theorem 8, e.g., by enabling low cost inductive proofs. This check tends to be significantly easier than the property check itself, as it merely attempts to validate that the modified constraint does not *alter* the hittability of the target along any trace, independently of whether the target is hittable or not. Additionally note that $init\_r$ can readily be eliminated using retiming.

13 Conclusion

We have discussed how various automated netlist transformations may be optimally applied while preserving constraint semantics, including *dead-end states*. We have additionally introduced fully-automated techniques for constraint elimination, introduction, and simplification. We have implemented each of these techniques in the IBM internal transformation-based verification tool *SixthSense*. The synergistic application of these techniques has been critical to the automated solution of many complex industrial verification problems with constraints, which we otherwise were unable to solve.

Due to the relative lack of availability of complex sequential netlists with constraints, we do not provide detailed experimental results. The only relevant benchmarks we are aware of are a subset of the IBM FV Benchmarks [21]. These constraints are purely sequential, thus preventing their optimal elimination through reparameterization alone. However, we were able to leverage transformations such as retiming and constraint simplification to enable reparameterization to optimally eliminate 2 of 2 constraints from IBM_03 and IBM_06; 3 of 4 from IBM_10; 5 of 8 from IBM_11; and 11 of 14 from IBM_24.

References

1. C. Pixley, "Integrating model checking into the semiconductor design flow," in *Electronic Systems Technology & Design*, 1999.
2. Accelera. PSL LRM, http://www.eda.org/vfv.
3. M. Kaufmann, A. Martin, and C. Pixley, "Design constraints in symbolic model checking," in *Computer-Aided Verification*, 1998.
4. Y. Hollander, M. Morley, and A. Noy, "The *e* language: A fresh separation of concerns," in *Technology of Object-Oriented Languages and Systems*, 2001.
5. P. Jain and G. Gopalakrishnan, "Efficient symbolic simulation-based verification using the parametric form of Boolean expressions," *IEEE Transactions on CAD*, April 1994.
6. M. D. Aagaard, R. B. Jones, and C.-J. H. Seger, "Formal verification using parametric representations of Boolean constraints," in *Design Automation Conference*, June 1999.
7. A. Kuehlmann and J. Baumgartner, "Transformation-based verification using generalized retiming," in *Computer-Aided Verification*, July 2001.
8. H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, "Scalable automated verification via expert-system guided transformations," in *FMCAD*, Nov. 2004.
9. A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification," *IEEE Transactions on CAD*, Dec. 2002.
10. J. Yuan, K. Shultz, C. Pixley, H. Miller, and A. Aziz, "Modeling design constraints and biasing in simulation using BDDs," in *ICCAD*, 1999.
11. J. Yuan, K. Albin, A. Aziz, and C. Pixley, "Constraint synthesis for environment modeling in functional verification," in *Design Automation Conference*, 2003.
12. H. Mony, J. Baumgartner, V. Paruthi, and R. Kanzelman, "Exploiting suspected redundancy without proving it," in *Design Automation Conference*, 2005.
13. C. Leiserson and J. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, 1991.
14. J. Baumgartner, A. Kuehlmann, and J. Abraham, "Property checking via structural analysis," in *Computer-Aided Verification*, July 2002.
15. J. Baumgartner and H. Mony, "Maximal input reduction of sequential netlists via synergistic reparameterization and localization strategies," in *CHARME*, Oct. 2005.
16. J. Baumgartner, T. Heyman, V. Singhal, and A. Aziz, "An abstraction algorithm for the verification of level-sensitive latch-based netlists," *Formal Methods in System Design*, (23) 2003.
17. J. Baumgartner, A. Tripp, A. Aziz, V. Singhal, and F. Andersen, "An abstraction algorithm for the verification of generalized C-slow designs," in *Computer-Aided Verification*, July 2000.
18. E. Clarke, A. Gupta, J. Kukula, and O. Strichman, "SAT based abstraction-refinement using ILP and machine learning techniques," in *Computer-Aided Verification*, July 2002.
19. K. L. McMillan and N. Amla, "Automatic abstraction without counterexamples," in *Tools and Algorithms for Construction and Analysis of Systems*, April 2004.
20. P. Bjesse and J. Kukula, "Using counter example guided abstraction refinement to find complex bugs," in *Design Automation and Test in Europe*, 2004.
21. IBM Formal Verification Benchmark Library. http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/fvbenchmarks.html.