Scalable Conditional Equivalence Checking: An Automated Invariant-Generation Based Approach

Jason Baumgartner[1] Hari Mony[1] Michael Case[1] Jun Sawada[2] Karen Yorav[3]
[1]IBM Systems & Technology Group [2]IBM Austin Research Laboratory [3]IBM Haifa Research Laboratory

***Abstract*—Sequential equivalence checking (SEC) technologies, capable of demonstrating the behavioral equivalence of two designs, have grown dramatically in capacity over the past decades. The ability to efficiently identify and leverage *internal equivalence points* to reduce the domain of the overall SEC problem is central to SEC scalability. However, *conditionally equivalent* designs – within which internal equivalence may not exist under sequential *observability don't care* conditions – are notoriously difficult for automated SEC tools. This paper constitutes one of the first attempts to advance the scalability of SEC for conditionally equivalent designs through automated invariant generation, which enables an inductive solution to an otherwise highly-noninductive problem. Through careful software engineering and various heuristics, this technique has been demonstrated capable of yielding orders of magnitude speedup on difficult industrial conditional SEC problems, in cases constituting *the only method* that we have found to achieve an automated solution.**

I. Introduction

Equivalence checking refers to the process of demonstrating the behavioral input-to-output equivalence of two designs. Numerous equivalence checking paradigms exist in practice. Combinational equivalence checking (CEC) is a framework where the state elements of two designs have 1:1 correlation. Instead of directly checking input-to-output equivalence, CEC frameworks *assume* that correlated state elements are equivalent, and demonstrate that outputs – as well as next-state functions of the correlated state elements – are equivalent. CEC frameworks thus avoid computationally-expensive sequential reasoning by decomposing the overall equivalence check into a set of combinational proof obligations [1]. Due to ease of use and scalability, CEC is pervasively used throughout the semiconductor industry – for example, to validate that logic synthesis preserves the behavior of the original design.

Sequential equivalence checking (SEC) is a generalization of CEC wherein the designs being equivalence checked may not have a 1:1 state element correlation [1]. The benefit of SEC over CEC is obvious: if a *sequential* transformation is performed across the designs being equivalence checked, CEC is no longer directly applicable – at least, not without substantial manual or restrictive methodological guidance. Such sequential transformations – e.g., retiming, state re-encoding, unreachable-state based optimizations, etc. – are commonly used in the design of high-performance circuits. Due to its generality, SEC generally requires analysis of the sequential behavior of the designs being equivalence checked – and thus comes with substantially greater computational expense. To be precise, CEC solves a set of subproblems in class NP, whereas SEC solves a more monolithic problem in class PSPACE [2].

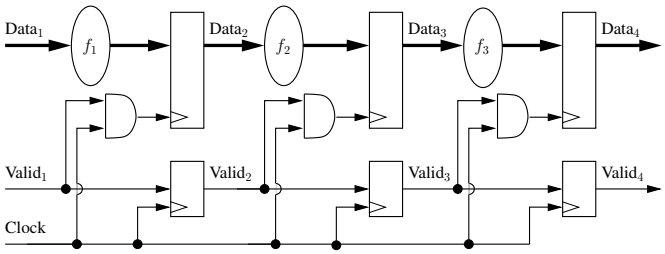

Fig. 1: Clock-gated pipeline

The ability to leverage internal equivalence points is often critical to the scalability of SEC. As with CEC, instead of merely demonstrating input-to-output equivalence, a set of internal equivalences may be demonstrated in conjunction. This overall set of properties is often substantially easier to solve than direct input-to-output equivalence: invariants that stipulate internal equivalences often enhance inductivity by strengthening the collective induction hypothesis [3], [4]. However, unlike CEC, it is generally not the case that every state element constitutes an internal equivalence point; the two designs may not have 1:1 correlation of state elements, nor behavioral equivalence among those which appear correlated (e.g., having identical signal names in their HDL definition).

Speculative reduction is often used to speed the overall SEC process. This technique consists of merging fanout references to suspected-equivalent gates even before they are proven equivalent, thereby simplifying proof goals expressed over this fanout logic. To ensure soundness, the validity of the speculatively-merged gates is checked as a set of additional proof obligations, and the SEC process attempts to solve all proof goals in conjunction. It has been demonstrated that the well-architected use of speculative reduction enables up to five orders of magnitude speedup on difficult industrial SEC problems, often making the difference on whether or not a conclusive result may be obtained [5], [6]. While SEC remains a problem in PSPACE as internal equivalence points may span arbitrary sequentially-redesigned subcircuits, speculative reduction often enables lighter-weight algorithms such as structural simplification and induction to solve problems which otherwise would require heavier-weight proof techniques.

In this paper, we address a generalization of SEC: *conditional sequential equivalence checking* (CSEC). Unlike the above-mentioned SEC paradigm, wherein equivalence is checked at all points in time and across all execution sequences, CSEC allows designs to depart from equivalent behavior under specific time-frames. For example, consider

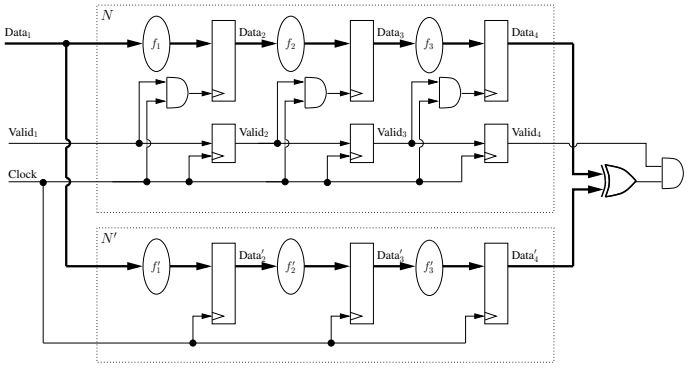

Fig. 2: Clock-gated CSEC testbench

the design of Figure 1 which pipelines its data computation across three clock periods. When a given pipeline stage is not required to produce a valid result, its clock may be disabled to reduce power consumption: a low-power technique called *clock gating* [7]. It is often desirable to validate that a version of the design with clock gating disabled (or not integrated) produces equivalent results to a version with clock gating enabled, to ensure that clock gating does not inadvertently alter design behavior. This is depicted in Figure 2, where N is equivalent to Figure 1 and N' is a simplified version without clock gating. Note that this is a CSEC problem: when no valid instruction is being processed ($\text{Valid}_i \equiv 0$), N will hold the results of its last computation whereas N' will needlessly process whatever invalid data is present at its inputs. The equivalence check over Data_4 vs. Data'_4 is thus restricted to conditions where valid output is required from N.

Power gating is a related CSEC problem domain: when a design component is idle, its voltage supply may be disabled by a controller. Power is restored only when that controller detects an imminent processing need [7]. Power-disabling is often modeled in verification by randomizing or "tristating" state element contents when their voltage is disabled [8].

CSEC problems are notoriously difficult to solve, since they lack the internal equivalences which are key to the scalability of traditional SEC approaches. While internal equivalences no longer unconditionally hold in CSEC problems, it is often the case for correct designs that internal equivalences hold *conditionally*. For example, in Figure 2, when $\text{Valid}_i \equiv 1$, the state elements of Data_i vs. Data'_i are pairwise equivalent. Conversely, the only time-frames at which corresponding state elements are inequivalent is an *observability don't care* (ODC) condition, which may never propagate to the outputs of the design at relevant time-frames. Given an adequate set of conditional equivalence invariants, it is often the case that the CSEC problem becomes k-inductive whereas otherwise it would not be. This observation is the motivation of this paper: to automate the derivation of an adequate set of invariants to in turn enable an efficient automated CSEC solution. We furthermore assume a synthesis-unaware paradigm for maximal applicability and generality, e.g., in case such optimizations are performed manually. We thus do not rely upon synthesis-history hints (e.g., [9]) for applicability of our techniques, though such hints may readily be incorporated into our framework to further enhance its scalability.

II. Preliminaries

The domain of the equivalence check is assumed to be over two hardware designs represented as *netlists*.

Definition 1: A *netlist* is a tuple $\langle\langle V, E\rangle, G, I, O, M\rangle$ comprising a finite directed graph with vertices V and edges $E \subseteq V \times V$. Function $G : V \mapsto types$ represents a mapping from vertices to $gate\ types$, including primary inputs $I \subseteq V$, state elements which each have an associated *initial-* and *next-state function*, and combinational gates with various functions.[1] Set $O \subseteq V$ represents the primary outputs, and set $M \subseteq V \setminus \{I, O\}$ represents the visible internal gates.

Netlists are often analyzed using *Boolean modeling*, wherein every gate evaluates to a Boolean value over time. Alternatively, *ternary modeling* may be used wherein gates take values from the set $\{0, 1, X\}$, where X refers to an "unknown" value. A gate that evaluates to a ternary X value is referred to as *tristated*.

Definition 2: An *equivalence checking testbench* comprises a netlist N'' which is the composition of two netlists N and N' under bijective mapping $I \mapsto I'$, along with possibly additional testbench logic; bijective mappings $O \mapsto O'$ and $M \mapsto M'$; and an *equivalence condition mapping* $C'' : O \mapsto V''$. The composition of N and N' is such that correlated elements of I and I' become merged as a single primary input. Mapping C'' defines the equivalence checking objectives for the testbench, i.e., a property $\big(C''(o) \rightarrow (o \equiv o')\big)$ to check of each correlated output pair $\langle o, o'\rangle$ of $\langle O, O'\rangle$.

Testbench logic refers to gates introduced to a netlist solely for verification purposes. E.g., any necessary input constraints may be synthesized and composed with the primary inputs of the netlist, or be enumerated with designated language constructs [10]. Similarly, logic may be introduced during synthesis of properties to be checked of the netlist, e.g., to specify equivalence conditions C''. For SEC, the range of C'' is simply $\{\top\}$: correlated outputs are checked for equivalence at all time-frames. For CSEC, C'' indicates at which points in time – and along which execution streams – output equivalence is to be checked (e.g., Valid_4 in Figure 2).

Note that C'' along with mappings $I \mapsto I'$ and $O \mapsto O'$ are *required* to establish the semantics of an equivalence checking testbench. Practically, mapping $M \mapsto M'$ is often readily available for the problem domains of interest since the designs being equivalence checked are often identical modulo the addition or enabling of extra circuitry, which is the optimization that induced the conditional equivalence. Examples include power gating and clock gating as discussed above, or general *sequential ODC*-based optimizations, wherein portions of the

[1] In this definition, we assume that gate behavior is independent of input permutation: for example, state elements with distinct *clock* and *data* inputs have been synthesized into simple one-input delay gates with adjacent combinational logic to preserve clocking semantics. This definition may be straight-forwardly extended to more general netlist types if desired.

designs may produce different results under conditions where those differences do not propagate to a failed equivalence check. We discuss the relevance of this mapping in the following section.

A. Invariants

An *invariant* is a property of a netlist which holds in all reachable states. An invariant may be represented through added circuitry as a gate which evaluates to $\top$ in every reachable state. While a functionally redundant characterization of netlist behavior, an invariant may be used to reduce the degree of overapproximation of certain proof techniques, and thus enable a more efficient solution. For example, k-induction is a proof framework that attempts to demonstrate that no state (reachable or not) which cannot violate a property within k time-frames may do so in greater than k time-frames [11]. The overapproximation inherent in induction is that if this particular check fails, one generally cannot determine whether the failing "inductive state" is actually reachable or not. Similarly, interpolation is a framework which overapproximates the reachability analysis of a netlist [12], risking the appearance that some property-violating unreachable states are actually reachable. Invariants may be used to enhance such frameworks because they constrain the overapproximation toward exact netlist behavior [4], [13].

Numerous techniques have been proposed for automated invariant generation. Examples include those which derive constant and equivalent gate pairs [3], [4]; those which derive implications among gates [4], [14]; those which attempt to enumerate invariants expressible using k-literal clauses [15], [16]; and heavier-weight techniques which perform approximate reachability analysis [13]. Some invariant learning approaches are *eager* in attempting to derive arbitrary invariants of a certain characterization [4], [15], [16], [14], while others are *lazy* in attempting to establish only invariants which will preclude spurious property failures in an overapproximating verification framework [17], [18]. A general tradeoff which must be considered is how much effort to devote to invariant generation vs. to spend directly attempting to solve the problem under consideration. Of even greater importance is the tailoring of the invariant generation process to the problem domain being addressed.

In this paper, we propose an eager framework for invariant generation tuned for enabling complex CSEC problems to be efficiently solved using overapproximating proof techniques such as induction and interpolation. In particular, given an equivalence checking netlist N'', our goal is to derive a set of conditional equivalence invariants of the form $\big(g_i \rightarrow (m_i \equiv m_i')\big)$ for some gate g_i and *correlated-gate pair*[2] $(m_i, m_i') \in M''$ to enable an efficient proof of the CSEC objectives. We refer to the set of gates g_i for which $(m_i \equiv m_i')$ as the *equivalence conditions* of m_i and m_i', denoted $E(m_i)$ or equivalently $E(m_i')$. Collectively, the set of derived invariants are $\big\{\forall (m_i, m_i') \in M''.\forall g_i \in E(m_i).\big(g_i \rightarrow (m_i \equiv m_i')\big)\big\}$.

[2]We hereafter abuse notation, referring to $M \mapsto M'$ as the relation M''.

Generally, one need not limit conditional equivalence invariants to this syntax: one may attempt to derive invariants over arbitrary gate triples, for example. However, such an approach is unattractive for industrially-relevant netlists because: **(1)** there are a *cubic* number of such invariants possible, and implication analysis – which entails only a *quadratic* number of candidate invariants – is known to face scalability challenges hence occasionally must be performed lossily [14]. The proposed syntax reduces the number of candidate invariants to quadratic. **(2)** As per the clock-gating and power-gating examples of Section I, this syntax is adequate to capture a suitably-expressive set of invariants to enable the efficient solution of otherwise highly-intractable CSEC problems. **(3)** We have developed numerous heuristics to expedite the eager derivation of candidate invariants of this form. In contrast to eager generation, we have found lazy generation of such invariants – e.g., based upon conditions witnessed from induction counterexamples [17] – to be of comparable or even greater computational expense due to the large number of candidate invariants of this form.

Additionally, note that the intrinsic purpose of our invariant generation framework is to cope with ODC-based optimizations across the netlists being equivalence checked. Despite a large amount of work in practical ODC derivation (e.g., [19], [20]), such approaches grow intractable in computational complexity as the logic windows against which ODC conditions are computed grow in size. The nature of the CSEC problem generally relies upon ODCs which span the majority of the netlist, thus rendering direct ODC computation to be an impractical approach to enabling the solution of CSEC problems, and motivating our invariant generation approach.

III. Algorithms

In this section we discuss our algorithms for computing conditional equivalence invariants. Our basic flow is depicted in Figure 3, and is similar to those used for traditional SEC, e.g. [3], [5]. In particular, we first compute a set of candidate invariants in Step 1, some of which may be invalid. In Step 2 we may use underapproximate analysis techniques such as random simulation and bounded model checking [21] to efficiently eliminate invalid candidates. We then iterate a proof phase attempting to validate the remaining invariants in Step 3. If any candidate invariants cannot be proven, due to being incorrect or due to being computationally intractable, those invariants are eliminated and another proof is attempted over those which remain in Step 4. The reason for iterating this check until all candidates are proven together is that it is often desirable to cross-leverage each invariant to tighten any overapproximate analysis used to prove the other invariants, e.g., to strengthen a collective induction hypothesis. Any invalid candidates may jeopardize the soundness of other proof results in such a framework.

There are two competing objectives in the choice of candidate invariants in Step 1. First: by considering each gate g in the equivalence checking netlist as a possible equivalence condition for each element of M'', a quadratic number of

1. Derive a set of candidate conditional equivalence invariants $E(m_i)$ for each $(m_i, m_i') \in M''$
2. Optionally utilize underapproximate analysis to falsify invalid invariants, pruning E
3. Attempt to prove that each of the candidate invariants is accurate
4. If any invariants cannot be proven, prune them from E; goto Step 3
5. All invariants have been proven accurate; return the resulting E

Fig. 3: Conditional equivalence invariant generation

invariants may be considered in this framework. For large netlists, direct consideration of such a large number of candidate invariants may require exorbitant resources. We address this concern in Section III-B. Second, the goal of this invariant generation process is to yield an adequate invariant set E to enable a subsequent efficient overapproximate proof technique – ideally induction – to solve the resulting CSEC problem. Approaches to subset the candidate invariants jeopardize the derivation of an adequate set of invariants for this purpose.

Note that merely considering every gate in the netlist as a possible equivalence condition is theoretically lossy in itself, in that a condition under which a particular conditional equivalence holds need not exist in the form of a gate in the netlist. We have found this a minor concern in practice. In applications such as clock gating and power gating, the necessary conditions often appear in the design in the form of gates that are added as an essential part of the optimization itself. In many other cases, the necessary conditions naturally appear in the synthesized testbench logic, since this logic specifies when output equivalences are to be checked. Finally, when ternary modeling is used, to model power gating, an undriven bus, an undefined *case* condition, or other *don't care* conditions [22], an often-adequate condition is to check equivalence whenever a correlated-gate pair is *not tristated*.

A. Computing Candidate Invariants

There are numerous problem domains where the number of candidate invariants of relevance may be contained to constant on a per correlated-gate basis. In particular, consider the use of ternary modeling wherein the X-value reflects the only conditions under which portions of the netlists being equivalence checked may differ. A linear number of candidate invariants may be derived using the procedure of Figure 4. Examples of such cases include power-gating verification (refer to Section IV-B), as well as cases where ternary X may be used to model don't care conditions or floating buses [22]. Another example is *uninitialized* sequential equivalence checking (e.g. *alignability analysis* [23]), which may be modeled by initializing all state elements with a ternary X, then driving a *weakly synchronizing input sequence* that brings the equivalence checking netlist into a state containing fewer X'es from which the outputs are pairwise equivalent and Boolean-valued [24].

In other problem domains, a potentially quadratic number of candidate invariants may need to be considered. In these cases, the practicality of our invariant generation framework is critically dependent upon the use of efficient algorithms and careful software engineering. Our solution has the following characteristics.

1. Build a ternary-modeled equivalence checking testbench
2. For every $(m_i, m_i') \in M''$, add candidate invariants into E of the form $\{(\neg tristated(m_i) \rightarrow (m_i \equiv m_i')), (\neg tristated(m_i') \rightarrow (m_i \equiv m_i'))\}$
3. Goto Step 2 of the algorithm of Figure 3 to obtain the final accurate E

Fig. 4: Ternary conditional equivalence candidate generation

(1) As illustrated in Figure 3, our basic flow is to first associate a set of candidate equivalence conditions with each gate pair in M'', then to use underapproximate analysis techniques to eliminate many of the invalid candidates, and finally to attempt to prove the remaining candidate invariants correct. We represent the candidate invariants using a *trie* data structure, which stores sets of candidate equivalence conditions for each $(m_i, m_i') \in M''$. The benefits of using a trie are that each unique equivalence condition set only requires a single data representation, and more importantly that common subsets of candidates across different correlation pairs may share in their data representation.

(2) When one counterexample trace is obtained (typically using incremental SAT) which invalidates one candidate invariant, we use a highly-tuned random simulation process to efficiently rule out large sets of invalid invariants, similar to what is done for traditional SAT-sweeping [25]. In particular, we use a *bit-parallel simulator* which models the behavior of the netlist across a variety of randomized input patterns, though seeded to adhere to the behavior witnessed in the original counterexample. By using 1024 parallel simulation patterns, and by extending the length of this random simulation by several time-frames with respect to the original counterexample length, we commonly witness ratios of 100:1 to 100000:1 in terms of the number of invalid candidates ruled out by re-simulation of SAT traces vs. the number of satisfiable SAT calls themselves. We have tuned the integration of the bit-parallel simulator with the trie data structure so that incorrect candidates are pruned as efficiently as possible.

Overall, while in the worst case a quadratic number of candidates may need to be considered, through careful software engineering we have reduced the memory overhead and runtime of our conditional equivalence invariant generation framework to approach linearity as closely as possible.

B. Subsetting Candidate Invariants

An additional motivation for efficiency is to attempt to minimize the number of possibly valid candidate invariants, while nonetheless ensuring that the resulting set preserves overall CSEC inductiveness. We have found the following heuristics useful for this purpose, which are selectively applicable on a per-problem-domain basis. Regarding which of these heuristics to select for a particular CSEC problem: note that information about the nature of the CSEC testbench may often yield insight into which of these heuristics (or the ternary conditional equivalence approach of Figure 4) may be most applicable. Additionally, note that in a fully-automated flow, these various algorithmic flavors may be run in parallel to the fully-quadratic algorithm, enabling convergence to a conclusive solution with a minimum of elapsed walltime. Automated sharing of invari-

1. for (i=0; ¬*terminate*; i++)
2. T_i = set of gates which may toggle at time i along any trace
3. M_i = gate pairs from M'' which may mismatch (or toggle from a mismatch to a match) at time i along any trace
4. Enumerate T_i as candidates for M_i in E
5. Validate E at time i
6. Goto Step 2 of the algo of Figure 3 to obtain the final accurate E

Fig. 5: Toggle-based equivalence candidate generation

ants across these various algorithms is also possible, which may incrementally approach the exhaustiveness of the fully-quadratic algorithm while benefitting from the speed of the subsetting techniques.

(1) One often may meaningfully subset the set of candidates on a per correlated-gate basis by analyzing the sequential behavior of the netlist. For example, considering the pipelined example of Figure 1, intuitively the logic comprising pipeline stage i would not constitute useful conditions for pipeline stages $j \neq i$. For general cyclic sequential netlists, the question of *which* gates may be effective conditions for *which* correlated-gate pairs becomes complex due to the inability to meaningfully structurally associate "pipeline stages." Our solution to this problem is to analyze *toggle* and *mismatch* activities of equivalence condition candidates and correlated-gate pairs, as depicted in Figure 5. Steps 2, 3, and 5 may generally be performed using a combination of incremental SAT and random simulation.

For certain types of designs, e.g. pipelines such as Figure 1, merely checking which gates may first toggle to a particular value at time i (set T_i), and associating them as candidate equivalence conditions for correlated-gate pairs which may first mismatch at time i (set M_i) is effective. In other cases, e.g. power-gated designs, more importantly than when correlated-gate pairs may first *mismatch* is when they may first transition from inequivalent to *equivalent*.

(2) Until a gate toggles for the first time, that gate is effectively constant hence is not useful as an equivalence condition for any correlated-gate pair. Similarly, until a pair of correlated gates mismatch for the first time, there is no need to attempt to learn invariants over them. Doing so would entail an unconditional equivalence invariant – or tautology – which may be handled more efficiently as per Section III-C.

(3) Often, many correlated-gate pairs share the same equivalence conditions. For example, each state element of a given pipeline stage in Figure 2 has the same Valid_i as an equivalence condition. It thus is often useful to assess the invalidity of a large set of candidate equivalence conditions for a small subset of correlated-gate pairs, e.g., several representatives identified at a given time-frame in Step 3 of the algorithm of Figure 5. Only those which appear valid for this subset will be considered for the remaining correlated-gate pairs.

(4) Structural analysis may be used to further prune candidates in cases, e.g., to only consider gates with substantial fanin overlap between a correlated-gate pair as equivalence condition candidates thereof. Such a technique often works well for clock-gating and power-gating verification, as the equivalence condition for a given state-element pair is often directly used to clock one of the two state elements. However, for more general sequential ODC-based optimizations, such structural prunings may fail to capture adequate conditions which may only be present in logic which flows around the redesigned subcircuits and/or be present at the testbench level alone. Nonetheless, in applicable cases such structural pruning may dramatically improve runtime: either limiting equivalence condition candidates to overlap with the fanin cone of the correlated-gate pair, or limiting conditions to be gates in the testbench logic.

(5) While we have not yet needed to rely upon such techniques, it is possible to use known invariant-reduction methods to minimize the cardinality of the candidate invariant set either in a semantically lossless or lossy manner. For example, *transitive reductions* may be used to reduce the number of implication-based invariants with no loss in their semantic power [14]. Just as invariants $(a \rightarrow b)$ and $(b \rightarrow c)$ subsume invariant $(a \rightarrow c)$, one may apply similar subsumption rules for conditional equivalence invariants: if $(a \rightarrow b)$, then $\big(a \rightarrow (m_i \equiv m_i')\big)$ subsumes $\big(b \rightarrow (m_i \equiv m_i')\big)$. Additionally, one may attempt lossy candidate invariant pruning using techniques such as ranking the relative constraining power of the candidates and retaining only a subset of greatest strength [16].

C. Incorporating Unconditional Equivalence Invariants

While the primary goal of our CSEC solution is to establish invariants over correlated-gate pairs which are conditionally equivalent (and thus conditionally *inequivalent*), in practice the netlists being equivalence checked may also contain a subset of correlated-gate pairs which are *unconditionally* equivalent. Such unconditional equivalence often must be considered to ensure inductiveness of the overall CSEC problem.

Merely using traditional SEC algorithms to prove-then-merge unconditionally-equivalent (or constant) gates, prior to application of our CSEC invariant-generation framework, may partially capture the unconditional equivalences. However, this approach is often insufficient since the presence of the conditionally-equivalent gates, without their corresponding conditional-equivalence invariants, renders the proof of unconditional equivalence intractable.

The most effective way that we have found to intermix the conditionally- and unconditionally-equivalent gate demonstration solutions is to nest the proposed conditional-equivalence invariant generation algorithm inside a traditional SEC framework using *speculative reduction* [5], [6]. Specifically, one may first postulate a set of candidate unconditional equivalences using a traditional SEC framework. Then, instead of directly proving those candidates, one may speculatively reduce the netlist such that fanout references to the candidate unconditional equivalences reflect a merge, thereby simplifying the overall set of proof obligations – while retaining a proof obligation to validate that the speculative merge candidates truly are equivalent. This speculatively-reduced model may be used as the basis of our CSEC framework, which will be used to prove the overall CSEC objectives in addition to the unconditional speculative-reduction proof obligations.

A comparable solution is to first generate candidate conditional-equivalence invariants, then to use a traditional SEC framework to prove those invariants in conjunction to any unconditional equivalences. This approach tends to be somewhat less efficient, however, as the CSEC framework needs to operate on a larger netlist, possibly managing more semantically-equivalent invariants.

D. Auxiliary Invariants

In addition to conditional and unconditional equivalence invariants, it is occasionally necessary to establish other types of invariants to ensure an overall inductive CSEC problem. For example, consider the CSEC testbench in Figure 2. This testbench includes two copies of the netlist of Figure 1 – N with an unconstrained Valid_1 to enable clock-gating, and N' with clock gating disabled. Additionally, some auxiliary testbench logic is often used to synthesize the equivalence condition under which output equivalence will be checked. For simplicity, in Figure 2 this condition is simply $Valid_4$ directly from N. However, due to the risk that the clock-gated design is improperly implemented, it is often desirable to use a correct-by-construction testbench-level equivalence condition.

It is sometimes necessary to derive invariants that establish a correlation between such testbench-level equivalence condition logic, and logic within the netlists being equivalence checked which indicates inactivity or *don't care* conditions. For example, in Figure 2, these invariants would simply correlate stages of the *valid* pipeline created at the testbench level as being equivalent to the corresponding $Valid_i$ signals in the clock-gated netlist. If such invariants are missing, k-induction may fail, confused by possible inductive starting states where the testbench logic indicates that output equivalence is to be checked, yet the gated netlist appears inactive hence conditional equivalence invariants do not apply.

When necessary, such invariants often may be established as a byproduct of an industrial-strength verification framework: either through lighter-weight structural simplifications (e.g., redundancy removal) commonly employed to reduce the size and complexity of a verification problem prior to invoking any costly decision procedures, or through the use of a coupled SEC framework to identify unconditional equivalences prior or subsequent to establishing conditional equivalence invariants as discussed in Section III-C.

The proposed invariant generation framework is generally incomplete in its goal of enabling an overall inductive or interpolation-amenable CSEC solution. While we have not encountered cases where such a framework was inadequate, we believe that – even if more elaborate invariants of this type were to be necessary – they would be straight-forwardly enumeratable by the engineer who set up the CSEC testbench given the need to have a basic understanding of how to specify the equivalence condition mapping C'' thereof. Additionally, we note that the proposed framework is readily incorporatable with other invariant generation and proof techniques. As our experiments demonstrate, we have found the proposed framework extremely powerful in enabling the automated solution of classes of difficult CSEC problems which we otherwise were not able to solve using any algorithms available to us.

E. Speculative Reduction over Conditional Equivalences

The concept of speculative reduction may be extended from unconditional equivalences – a fanout-merge – to conditional equivalences by injecting a multiplexor. Specifically, given $(m_i, m_i') \in M''$, one may replace fanout references to m_i' by the function *if* $(\bigvee_{g_i \in E(m_i)} g_i)$ *then* (m_i) *else* (m_i'). However, unlike speculative reduction for unconditional merges which is guaranteed not to increase the size of the resulting netlist, speculative reduction for conditional merges is almost guaranteed to increase netlist size. We thus have not yet experimented with the use of this technique, though postulate that it may benefit various algorithms in structurally reflecting the ODC condition implicit in the set of conditional equivalence invariants.

IV. Experimental Results

In this section we provide experimental results to illustrate the benefits of our conditional equivalence invariant generation techniques. The set of CSEC problems that we have encountered to date is admittedly somewhat small: there are no public CSEC benchmarks available to our knowledge (most are direct SEC problems), and the majority of industrial problems also fall into the CEC or SEC category: even if they contain sequential ODC-style optimizations, those optimizations may often be verified using a highly-tuned multi-algorithm SEC toolset to cope with any noninductive subcircuits (e.g., [5], [6]). We nonetheless have found this invariant generation framework useful to enable efficient inductive solutions to numerous of these type of problems, using whichever heuristics discussed in Sections III-A and III-B apply to the particular problem under consideration.

There are however numerous CSEC problem domains where even the best-tuned SEC solution simply cannot scale, such as power-gating and globally-optimal clock-gating. This work has been motivated by the existence of such difficult CSEC examples which we have been unable to solve without substantial manual decomposition or abstraction, and the knowledge that such problem domains will increase in prevalence as the demand for low-power devices continues to rise. Furthermore, as the sophistication of synthesis tools continues to grow, their ability to leverage more intricate ODC conditions will also continue to grow; equivalence checking tends to be less scalable than synthesis itself, since the algorithms in the former generally need to reason about a design twice as large as the latter. We thus focus our experimental discussion on one representative example from each of these categories of difficult CSEC problems that otherwise were unsolvable using existing algorithms. In these examples, the gates in M'' are restricted to state elements. Our invariant generation framework has been implemented in the IBM internal verification tool *SixthSense* [26], [5], [10], [6], [27].

A. Case Study 1: Clock Gating

This example is a high-performance double-precision floating-point unit (FPU) with fused multiply-add capability. It contains a 53x54 multiplier, adders, shifters, a leading-zero anticipator, and other logic components typically associated with FPUs. It has a pipeline of 12 clock periods due to its high clock frequency. This FPU is designed to operate configurably with or without clock-gating enabled; this testbench validates the conditional equivalence of these two operational modes.

The issuing of an instruction into the FPU pipeline is indicated by the assertion of an *Issue* primary input. Internally, this input is pipelined, and this pipeline is used to indicate whether there is a valid instruction at a particular stage so that its clock is enabled (similar to the Valid_i pipeline of Figure 1). At the testbench level, *Issue*s are also pipelined and a conditional output equivalence check is associated with this testbench-level pipeline.

The complexity of the arithmetic logic, and the high degree of bit-level optimization thereof, make this example particularly challenging. The design itself comprises more than 23k lines of RT-level VHDL, which does not include several synthesized components. This complexity prevents us from performing even bounded model checking of a single instruction through the FPU within 24 hours. The only way that we have been able to solve this CSEC problem in the past, before the automated conditional equivalence invariant generation solution was available, has been through time-consuming manual abstraction of the arithmetic subfunctions between pipeline stages using uninterpreted functions, then brute-force automated abstraction and reachability algorithms.

This CSEC netlist has 21,453 state elements, 542 inputs, and 130,412 And gates. We limited conditional equivalence candidates to be those appearing at the testbench level alone, due to the creation of the above-mentioned correct-by-construction pipeline indicating when output equivalence is to be checked. The goal of the conditional equivalence checking invariant generation framework is thus to identify which stages of the testbench pipeline imply conditional equivalence of which correlated state elements of the two versions of the FPU. There are 254,016 candidate invariants using this approach. Of these, bounded model checking falsified 357, whereas our bit-parallel simulator falsified 242,330: a ratio of 1:679. The remaining 11,329 invariants were proven valid using 3-step induction, from which the overall CSEC objectives became 1-step inductive. The number of gates participating in any conditional equivalence condition set E for these invariants was 16. The resources required for this run are 14,459 seconds on a 2.16 GHz processor, with 2.0 GB peak memory.

B. Case Study 2: Power Gating

Our next case study demonstrates the applicability of our conditional equivalence invariant generation framework to solve difficult power-gating CSEC problems. This testbench is the result of applying the power-gating verification methodology presented in [8]. The design being equivalence checked is a four-port out-of-order execution unit, which can compute loads, stores, adds, subtracts, shifts, and branches on 32-bit data, manipulating a 16-entry register file. This CSEC testbench includes two copies of the execution unit: one with power-gating enabled, and one with power-gating disabled. This design comprises more than 13k lines of RT-level VHDL.

The verification setup of [8] performs SEC under conditions imposed by an *observer*. The observer is a testbench-level circuit that observes the netlists being equivalence checked, and raises a flag when the execution unit is active (i.e., is working on some command). This flag constitutes the equivalence condition gate g''_o under which each output o is equivalence checked. This flag also reflects the violation of environment-assumption violations; g''_o forever deasserts after such a violation, trivializing the equivalence check. This example is an extended version of the one reported in [8], which is substantially more challenging to solve. *All previous attempts to solve this CSEC problem failed, even when given virtually unlimited memory and time limitations, using any available algorithms*. Even the manually-simplified variant of this problem reported in [8] required 91 hours of runtime.

This example used the algorithm of Figure 4, establishing that when not tristated, correlated state-element pairs are equivalent. Additionally, we added candidate invariants checking whether each state element may ever be tristated to further strengthen the induction hypothesis; similar strengthening could be achieved by nesting the invariant-generation framework within a SEC framework as discussed in Section III-C. We used an automated technique to safely learn *verification constraints* [28], correlating to the observer having witnessed assumption violations which forever thereafter entails a trivial pass of the equivalence check. These constraints enhanced the inductivity of the CSEC objectives, as uninteresting behavioral differences between the two designs became unreachable. We additionally leveraged automated phase abstraction [29] to temporally compress the clock period from two to one verification time-step, as well as an automated temporal shifting algorithm to eliminate the reset phase from the testbench [27], to collectively enhance inductiveness of the CSEC objectives.

This CSEC netlist has 807 state elements, 175 inputs, 22,326 And gates, and 417 outputs to be equivalence checked. Of 1,196 candidate invariants, 961 were proven valid using 3-step induction. The number of gates participating in any conditional equivalence condition set E for these invariants was 669. These invariants enabled all outputs to be proven equivalent with 1-step induction, with a total runtime of 149 seconds and 97.2 MB on a 2.16 GHz processor.

V. Conclusion

In this paper we have proposed an eager conditional equivalence invariant generation framework to enable the solution of difficult *conditional sequential equivalence checking* (CSEC) problems. Such problems, which commonly arise due to design optimizations such as clock gating and power gating, render traditional SEC frameworks highly unscalable due to the absence of internal unconditionally-equivalent

points. Nonetheless, often there are numerous internal points which are only inequivalent when such inequivalence is an *observability don't care*. Our invariant generation framework attempts to identify an adequate set of conditional equivalence invariants to render an efficient overall CSEC solution through scalable proof techniques such as induction. Our experiments demonstrate the power of this approach, enabling the efficient automated solution of several classes of problems which we otherwise were unable to solve using any available algorithms.

REFERENCES

[1] A. Kuehlmann and C. A. van Eijk, *Combinational and Sequential Equivalence Checking*. Kluwer Academic Publisher, 2001.

[2] J.-H. Jiang and R. Brayton, "Retiming and resynthesis: A complexity perspective," in *TCAD*, vol. 25, Dec. 2006.

[3] C. A. J. van Eijk, "Sequential equivalence checking without state space traversal," in *DATE*, Feb. 1998.

[4] P. Bjesse and K. Claessen, "SAT-based verification without state space traversal," in *FMCAD*, Nov. 2000.

[5] H. Mony, J. Baumgartner, V. Paruthi, and R. Kanzelman, "Exploiting suspected redundancy without proving it," in *DAC*, June 2005.

[6] H. Mony, J. Baumgartner, A. Mishchenko, and R. Brayton, "Speculative-reduction based scalable redundancy identification," in *DATE*, Apr. 2009.

[7] M. Keating, D. Flynn, R. Aitken, A. Gibbons, and K. Shi, *Low Power Methodology Manual*. Springer, 2007.

[8] C. Eisner, A. Nahir, and K. Yorav, "Functional verification of power gated designs by compositional reasoning," in *CAV*, July 2008.

[9] R. Brayton and A. Mishchenko, "Recording synthesis history for sequential verification," in *FMCAD*, Nov. 2008.

[10] J. Baumgartner, H. Mony, and A. Aziz, "Optimal constraint-preserving netlist simplification," in *FMCAD*, Nov. 2008.

[11] M. Sheeran, S. Singh, and G. Stalmarck, "Checking safety properties using induction and a SAT-solver," in *FMCAD*, Nov. 2000.

[12] K. McMillan, "Interpolation and SAT-based model checking," in *CAV*, July 2003.

[13] G. Cabodi, S. Nocco, and S. Quer, "Strengthening model checking techniques with inductive invariants," *TCAD*, vol. 28, Jan. 2009.

[14] M. Case, A. Mishchenko, and R. K. Brayton, "Inductively finding a reachable state space over-approximation," in *IWLS*, 2006.

[15] K. McMillan, "Don't care computation using k-clause approximation," in *IWLS*, 2005.

[16] M. Case, A. Mishchenko, and R. Brayton, "Cut-based inductive invariant computation," in *IWLS*, 2008.

[17] L. de Moura, H. Rueß, and M. Sorea, "Bounded model checking and induction: From refutation to verification," in *CAV*, July 2003.

[18] M. Case, A. Mishchenko, and R. Brayton, "Automated extraction of inductive invariants to aid model checking," in *FMCAD*, Nov. 2007.

[19] A. Mishchenko and R. Brayton, "SAT-based complete don't-care computation for network optimization," in *DATE*, March 2005.

[20] Q. Zhu, N. Kitchen, A. Kuehlmann, and A. Sangiovanni-Vincentelli, "SAT sweeping with local observability don't-cares," in *DAC*, July 2006.

[21] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *TACAS*, March 1999.

[22] M. Turpin, "The dangers of living with an X," in *SNUG*, Oct. 2003.

[23] C. Pixley, "A theory and implementation of sequential hardware equivalence," in *TCAD*, Dec. 1992.

[24] Z. Khasidashvili and Z. Hanna, "SAT-based methods for sequential hardware equivalence verication without synchronization," in *BMC*, 2003.

[25] A. Kuehlmann, "Dynamic transition relation simplification for bounded property checking," in *ICCAD*, Nov. 2004.

[26] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, "Scalable automated verification via expert-system guided transformations," in *FMCAD*, Nov. 2004.

[27] M. Case, H. Mony, J. Baumgartner, and R. Kanzelman, "Enhanced verification through temporal decomposition," in *FMCAD*, Nov. 2009.

[28] G. Cabodi, P. Camurati, L. Garcia, M. Murciano, S. Nocco, and S. Quer, "Speeding up model checking exploiting explicit and hidden verification constraints," in *DATE*, April 2009.

[29] P. Bjesse and J. Kukula, "Automatic generalized phase abstraction for formal verification," in *ICCAD*, Nov. 2005.