Scalable Sequential Equivalence Checking across Arbitrary Design Transformations

Jason Baumgartner[1] Hari Mony[1] Viresh Paruthi[1] Robert Kanzelman[1] Geert Janssen[2]
[1]IBM Systems & Technology Group[1] [2]IBM Research Division

Abstract

High-end hardware design flows mandate a variety of sequential transformations to address needs such as performance, power, post-silicon debug and test. Industrial demand for robust sequential equivalence checking (SEC) solutions is thus becoming increasingly prevalent. In this paper, we discuss the role of SEC within IBM. We motivate the need for a highly-automated scalable solution, which is robust against a variety of design transformations – including those that alter initialization sequences. This motivation has caused us to embrace the paradigm of SEC with respect to designated initial states. We furthermore describe the diverse set of algorithms comprised within our SEC framework, which we have found necessary for the automated solution of the most complex SEC problems. Finally, we provide several experiments illustrating the necessity of our diverse algorithm flow to efficiently solve difficult SEC problems involving a variety of design transformations.

I. Introduction

Combinational equivalence checking (CEC) is a framework commonly used for validating that logic synthesis does not alter the functionality of a design. For such applications, CEC operates on two versions of a design: the first is pre-synthesis, often a register-transfer level (RTL) version of a design which is used for functional verification; the second is post-synthesis, often a gate- or transistor-level version of the design. CEC operates by correlating primary inputs and latch points between the two designs, and proving that this pairing guarantees equivalence of all primary outputs and next-state functions. Due to ease of use and scalability, CEC has become by far the most pervasively-used form of formal verification throughout the industry. Many design methodologies enforce the use of CEC, which requires sequential transformations to be back-translated into the pre-synthesis RTL model [1].

While powerful, CEC is for the most part limited in applicability to designs with 1:1 state element pairings. CEC tends to become ineffective if significant sequential transformations are performed on a design, e.g., by optimizations such as retiming or FSM re-encoding, addition of power-saving logic such as clock-gating, etc. Due to the growing demands of hardware design, however, such transformations tend to comprise an increasing portion of the modifications performed during the life-cycle of a product. With a purely CEC-based methodology, this means that sequential transformations often require a full regression of the functional verification process, which is often time-consuming and, if simulation-based, incomplete. Practically, this means that performing sequential optimizations becomes somewhat of a bottleneck in design cycles, and that certain design suboptimalities are instead tolerated to avoid the risk of late-introduced bugs.

Sequential equivalence checking (SEC) is a paradigm to help offset these limitations of CEC. SEC performs a true sequential check of input/output equivalence, hence is not limited to operation on designs with 1:1 state element pairings. The benefits of SEC are manifold. For example, one may use SEC to efficiently prove the "correctness" of sequential transformations that preserve design functionality, without a need to re-run lengthier, often lossy functional verification regressions. This enables resource savings during the design cycle, and *eliminates the risk associated with sequential transformations* – enabling more aggressive optimizations than otherwise would be tolerated, especially late in the design phase. Ultimately, a methodology based upon SEC implies that sequential transformations can be automated by synthesis tools, allowing more abstract of reference models to be the basis of verification and synthesis.

In practice, SEC enables a more flexible set of applications than direct input/output equivalence, including white-box functional checks. For example, one may check the

1-4244-9707-X/06/$20.00 ©2006 IEEE

equivalence of specific *modes of operation* of a design, such as backward-compatibility modes of design evolutions. In cases, one may wish to deploy SEC against designs that are not even strictly equivalent – but can be made so by disabling initialization/test/debug logic, by ignoring output mismatches during *don't care* time-frames, by accounting for differing pipeline stages, etc.

Unfortunately, this benefit does not come without a price; SEC is much more computationally expensive than CEC. CEC often assumes not only 1:1 latch correspondence, but also 1:1 design hierarchy equivalence, enabling CEC to scale efficiently up to even the largest chip-level designs. SEC, in contrast, often does not assume latch or hierarchy equivalence; SEC certainly can benefit from such equivalences if they exist, but these relaxations are precisely the benefits that SEC is intended to offer over CEC. This often limits SEC in applicability to only smaller design units, mandating in cases a fair amount of user sophistication to decompose a larger design and specify invariants at the corresponding boundaries to yield a higher-level equivalence proof [2]. In many cases, the lack of scalability of SEC results in an incomplete application of the technology.

In this paper, we describe the use of SEC within IBM. Though SEC has also been used on various ASICs, our focus is primarily on IBM's high-end custom processor designs. The demands of such designs is quite extreme in all aspects, from aggressive clock periods that mandate timing-aware placement of level-sensitive latches, to circuit placement issues that may alter the amount of sequential redundancy in the design to minimize propagation delays, to aggressive power-saving needs that require techniques such as clock-gating, to elaborate initialization, test, and debug logic intertwined with the functional logic at all hierarchy points. Due to the need for the highest levels of performance, these optimizations are performed largely by hand. Thus, for the most part, SEC at IBM is currently used as an HDL-to-HDL pre-synthesis methodology, with CEC used to close the gap with post-synthesis gate- and transistor-level netlists [3].

This methodology has placed significant demands on our SEC solution, as it tends to be deployed against arbitrary transformations (even those that do not strictly preserve equivalence), without the ability to leverage any feedback from synthesis tools to simplify the check. Furthermore, this often results in our SEC solution being deployed at fairly large design slices to ensure that all changes comprising a non-local transformation are captured.

We discuss our SEC framework in Section II. Our SEC solution is comprised in *SixthSense*, which is an IBM internal (semi-)formal verification toolset for functional verification as well as for SEC. We discuss front-end and back-end aspects of SixthSense in Section III. In Section IV we discuss the technologies comprised within SixthSense, which have enabled us to scale SEC up to tens of thousands of state elements and beyond, across a variety of sequential transformations. In Section V we discuss some SEC experiments. Finally, in Section VI, we conclude this paper.

II. SEC with Designated Initial States

We represent the designs being equivalence checked as netlists, comprising gates of various types including constants, primary inputs, registers, and combinational gates with various functions. Registers have designated *next-state functions* whose time-*i* values define their behavior at time $i+1$. Registers also have (possibly symbolic) *initial values* which define their time-0 behavior.[1] A *state* is a cross-product of concurrent valuations to registers; an *initial state* is one producible at time 0. A *trace* is a set of $0,1$ valuations to a netlist's gates over time (or $0,1,X$ in ternary analysis).

Certain gates may be labeled as *targets*, where the verification goal is to obtain a trace illustrating an assertion of a target from a valid initial state, or to prove that no such trace exists. Other gates may be labeled as *constraints*, where a valid trace is one that assigns all the constraint gates to 1 for its possibly finite duration. Finally, certain gates may be attributed as *primary outputs*.

Sequential equivalence checking frameworks generally validate that from some cross-products of states, two designs exhibit identical sequences of valuations to their primary outputs under all possible sequences of valuations to their primary inputs. Given two netlists N_1 and N_2, such frameworks typically operate upon a composite netlist N_3 formed as the union of N_1 and N_2, though merging corresponding primary input gates to ensure that an identical sequence of input valuations is applied to both N_1 and N_2, and adding targets representing the exclusive-OR of corresponding primary output gates of N_1 and N_2.

Numerous notions of sequential equivalence have been proposed over the years. In coarse terms, these approaches can be categorized into two classes: those that prove equivalence with respect to a specified initial state set (e.g., [4]), and those that attempt to demonstrate resetability across all states during an equivalence proof (e.g., [5], [6], [7]). Our framework utilizes the former approach [8]. We have several compelling reasons for this choice.

First, initialization data tends to be readily available in some form for IBM designs. Once the design is mature, its initialization logic is integrated, and its initialization

[1] Time is thus defined on $\mathbb{N}$. Registers are presented as our only sequential gate type merely for simplicity of exposition; more complex state elements may be decomposed into registers and combinational gates.

sequence is known. For example, many designs use a scan-based initialization approach where a stream of configurable data is flushed through a serially-connected *scan chain* interconnecting most of the state elements of the design. The set of initial states, often referred to as the "power-on reset state," is obtainable via ternary simulation by X'ing all initial values; simulating the deterministic reset sequence (e.g., enabling the scan clock and providing the proper scanned-in data) and X'ing all other inputs; then using the resulting $0, 1, X$ state as the initial state [9].[2] Even when the design is less mature and the initialization logic is not yet implemented, an approximated initial state is available and designed against, often 0'ing all initial values aside from a small subset that is desired to behave otherwise. While the development of verification techniques that can be employed *before* initialization data is available are well-motivated (e.g., [7], [2]), this initial state requirement has not been a practical hindrance in our environment.

Second, this power-on reset state is the basis of functional verification as well as SEC at IBM. Ternary functional verification may be used to validate that initial-state X'es do not persist in the design, e.g., due to a possible deadlock state. Binary functional verification will randomize any X'es in the power-on reset state, ensuring that the design cannot violate any of its properties across any possible initial state resolution. SEC (using ternary *or*, more commonly, binary analysis) is also performed using this power-on reset state to help ensure that design modifications intended to preserve sequential equivalence do not jeopardize the truth of any properties. In this way, a successful sequential equivalence check can forgo the need to re-run possibly lengthy functional verification regressions.

Third, it is generally computationally easier to solve the problem of SEC given designated initial states vs. using alignability style of analysis. Though techniques for the latter have progressed substantially, e.g., through the use of SAT-based analysis [7], such approaches tend to be limited in applicability to designs with hundreds of state elements, or in cases, to a few thousand. In contrast, through the ability to leverage a robust set of transformation and verification algorithms, techniques for performing SEC against known initial states can readily scale up to designs with 10,000s of state elements [8].

Such scalability does rely to some extent on a significant number of internal equivalences, hence in cases a manual decomposition can be performed even under a more general equivalence definition, e.g., using the framework of [2]. However, we have found that relying heavily on manual decompositions is somewhat of a barrier to wider-scale adoption of SEC. In particular, such decomposition requires a significant amount of manual effort and sophistication to set up lower-level constraints to enable the equivalence check, whereas such constraints are inherent in the context of a larger design unit, automatically derivable modulo the complexity of the sequential analysis required to do so. We thus have spent considerable effort in scaling our SEC solution to as large of design slices as possible to enable a maximal return on investment from using the tool, relying upon manual decomposition only when absolutely necessary.

Fourth, for some of the design transformations against which we care to prove equivalence, traditional alignability-style analysis is inapplicable. Recall that our SEC framework operates largely against manual transformations of the design. Using scan-based initialization schemes, for example, even optimizations such as retiming may alter the necessary initialization mechanism to ensure equivalent functionality of a design. This may preclude a common initialization sequence for the designs being equivalence checked, even though independent initialization mechanisms do render the designs into a state from which they are *functionally* equivalent. In cases, it is desired to run SEC before initialization logic is even integrated into a design. Some of our applications furthermore require specific initial states, e.g., to test against specific *modes of operation* of a design; and some rely upon the addition of logic to the design itself to normalize out known mismatch conditions, as will be discussed in Section III. Such applications mandate the need for precise control of the initial state against which SEC will be performed, decoupling the problem of validating the initialization logic as a distinct verification task.

III. SEC Interface

In this section we discuss the front-end and back-end interface to our SEC toolset. We will keep this discussion brief, since the details of these interfaces are for the most part intuitive, if not trivial.

Our SEC toolset takes as input two gate-level design representations, typically obtained by a compilation of HDL using a light-weight synthesis process common with functional verification front-ends. By default, the two designs have their primary inputs and outputs correlated by simple name comparison, though this process can be overridden. Hooks are also available to equivalence check internal signals referenced in the functional verification environment, such that a successful SEC result

[2] Deterministic values result at most state elements through this simulation. Some may retain an X state either because they are not initialized during the sequence, or due to the weakness of ternary simulation. The latter tends to be a minor point since most initialization mechanisms are robust against the weakness of ternary evaluation, and due to methodology requirements the initialization logic will often be enhanced in robustness by the designer if an undesired X results.

can completely forgo functional verification regressions. Synthesizable properties may also be equivalence-checked, which allows checking arbitrary relations between the two designs.

As discussed in Section II, the desired initialization is typically the power-on reset state. This state is often available in a format which is usable for functional verification, and can be reused for SEC. Other possible default initialization options include the all-0 state or the all-random state, the latter of which often requires some constraints to prevent bogus mismatches. These defaults can be overridden on a fine-grained basis. For example, one may wish to override certain *mode* bits of a design, so that the SEC will check equivalence of a design across all hardware modes to be supported in a product, not just its default one. As another example, one may wish to validate that a non-default *backward-compatibility* mode of a redesign truly is sequentially equivalent to the prior version of that design, despite a great deal of sequential modifications.

Practically, additional hooks are necessary in SEC. First, one may need to constrain the input valuations of the design to avoid “uninteresting” mismatches. For example, often it is desired to check sequential equivalence only across the functional mode of the design. One will thus disable initialization logic and establish functional clocking during SEC, e.g., to prevent mismatches due merely to differing scan chain lengths. Other examples include cases where a design was optimized against an illegal input valuation (e.g., an invalid opcode); such valuations must be disallowed to avoid uninteresting mismatches. Such input constraints are conservative given the stimulus the design will be subjected to during functional operation, which should be independently validated for completeness.

Second, in cases, direct comparison of outputs may yield uninteresting mismatches even if input constraining is performed. For example, a redesign may add a pipeline stage to a design. Without accounting for this latency difference, output mismatches would be inevitable. As another example, one may optimize the design such that certain outputs may behave differently during *don’t care* time-frames. Clock gating is one prevalent case of this; when a design is idle, its clocks may be disabled to save power; however, the pre-clock-gated variant of that design may produce differing output valuations during such idle conditions. One may thus need to restrict the equivalence check to *care* time-frames.

Our SEC toolset supports an HDL-based language to allow a flexible mechanism for adding constraints, overriding primary inputs and internals, and injecting logic to be equivalence checked in place of certain outputs.

Given the two designs, their initialization data, and any optional constraints, overrides, or alternate controls such as black-boxing, the SEC is ready to run. The outcome of this run is, on a per-compare-point basis, either a proof of equivalence; the demonstration of a mismatch; or an unsolved “partial validation” result indicating information such as the number of time-frames from the initial states for which a mismatch provably cannot occur. Mismatches are reported in the form of traces, which can be viewed using an IBM-internal waveform / design source browser, illustrating valuations over time for both designs which lead to the mismatch. An “unsolved” result is uncommon even on large designs, and can almost always be resolved through hand-tuning the algorithm flow used by SixthSense (refer to Section IV-B). Otherwise, techniques such as black-boxing of large arrays, or cutpointing temporally-deep logic to facilitate the use of semi-formal approaches to more adeptly identify redundancies in the composite netlist, may be used.

IV. SEC Algorithms

In this section we discuss the algorithms used in our SEC toolset. SixthSense is a multi-algorithmic *transformation-based verification* (*TBV*) tool. TBV is a framework proposed in [10] wherein one may synergistically utilize various transformation algorithms to iteratively simplify and decompose complex problems until they become tractable for automated formal verification. All algorithms are encapsulated as *engines*, each interfacing via a common modular API. Each engine receives a verification problem represented as a netlist, then operates on that problem to attempt to solve it (e.g., as with a reachability engine) or to attempt to simplify or decompose it (e.g., as with a retiming engine). In the latter case, it is generally desirable to pass the simplified problem to another engine to further process that problem. As verification results are obtained on the simplified problem, those results propagate through the sequence of engines in reverse order, with each transformation engine undoing the effects of the transformations it performed to present its parent engine with results that are consistent with the netlist that its parent transmitted.

SixthSense was initially developed for functional verification, hence our original set of engines was developed with property checking applications in mind. To extend the applicability of SixthSense to SEC, we added various usability hooks (refer to Section III), as well as targeted algorithms – primarily for sequential redundancy elimination. This has proven to be a synergistic framework, as we have found that all of the engines that we initially developed for functional verification have played a major role in enabling the solution of complex SEC problems, and vice versa.

1. Guess the *redundancy candidates*: equivalence classes of gates, where each pair of gates in an equiv class is suspected to be functionally equivalent modulo inversion.
2. Select a *representative gate* from each equiv class.
3. Construct the *speculatively-reduced model*: replace every fanin reference to a gate by a reference to its representative; add a *miter* over each gate and its representative.
4. Attempt to prove that each miter is unreachable.
5. If any miter cannot be proven unreachable, refine the equiv classes to separate the corresponding gates; go to Step 2.
6. All miters have been proven unreachable; the equiv classes reflect true redundancy hence their gates may be merged.

Fig. 1: Generic redundancy removal algorithm

A. Sequential Redundancy Removal

The technologies of our SEC tool have been presented in [8]. Our sequential redundancy elimination flow is as per Figure 1, using a scheme similar to that of [4], [11]. As noted in [4], "it is necessary to combine the detection and utilization of similarities to really benefit from them" during SEC proofs. This is accomplished via an *assume-then-prove* paradigm. This paradigm often begins with a redundant gate guessing approach as in Step 1, using a variety of techniques such as semi-formal analysis (random simulation and bounded model checking), structural analysis, name and hierarchy comparisons, etc. Note that semantic analysis is generally necessary for optimality; e.g., in case sequential redundancy was added to one design, *N*:*M* (vs. 1:1) register grouping may be needed.

Once the redundancy candidates are guessed, a speculative merge of the redundancy candidates is performed in Step 3 to create the model to be equivalence checked (the "assume" step). Finally, proof analysis is performed on the speculatively-reduced model to attempt to validate the correctness of the candidates (the "prove" step). This correctness is represented by the unreachability of a *miter*, which is an internally-generated target checking the exclusive-OR of two candidates. Failed proofs, whether falsified or inconclusive (e.g., due to an ineffective algorithm or insufficient resources), cause a refinement of the candidates and another assume-then-prove iteration.

In order to optimally leverage redundancies within the design, we have found that redundancy must be exploited not only at the register level, but also at the gate level. For example, retiming is a fairly global design optimization that may potentially relocate *all* registers in the design, rendering an assume-then-prove framework which only seeks redundancies across registers non-scalable. However, using gate-level equivalence-classing, coupled with an adequately strong proof technique, we can often efficiently solve the most complex of retiming-and-resynthesis problems as demonstrated in [8].

To avoid an excessive number of refinement loops in the algorithm of Figure 1, it is important to obtain a valid set of redundancy candidates from which the speculatively-reduced model is constructed. This is especially true for some of the larger designs that we have equivalence checked, which may have millions of gates and hundreds of thousands of registers; poor equivalence classing may be fatal to the conclusiveness of this algorithm. If pairing based upon semi-formal analysis is inadequate, one approach we have found to help augment this analysis is to perform miter falsification on the speculatively-reduced model prior to proof iterations [8]. This allows us to exploit the speculative mergings to enable deeper symbolic analysis, even if some of those speculative mergings are incorrect at deeper time-frames. We can also deploy light-weight algorithms before the core falsification effort, which may discard miters that are implied by others via low-cost transformation and proof analysis to better focus differentiation resources on the *root* incorrect pairings.

B. Scalable SEC via Algorithmic Synergies

What is particularly novel in our framework vs. prior approaches is the much richer set of algorithms that we use in the "prove" step. As we describe in [8], discarding redundancy candidates during refinement effectively weakens the induction hypothesis of the assume-then-prove framework. Discarding candidates due to ineffective proof techniques thus precludes the leveraging of those internal equivalences to simplify the proof of other miters after the refinement, which often ultimately avalanches into a failed proof of output equivalence. Simply stated, even within a single design component, every miter is essentially a distinct verification problem. It is well-known that using the proper set of algorithms for a given verification problem can be exponentially faster than using an inferior set. Prior work has relied primarily upon induction for the proof step [4], [12], possibly augmented with localized reachability analysis for redesigned regions which cannot be well-paired [11]. While such algorithms are indeed key in our framework, we additionally may leverage a variety of synergistic transformation and verification engines to attempt to discharge the miters. Our TBV framework can thus be seen as a robust and flexible *extension* of induction-based frameworks. In practice, we have found that this flexibility is critical to the solution of the most complex SEC problems.

Some of the engines that we have found particularly powerful in SEC proofs include the following.

- **COM**: a redundancy removal engine which uses combinational techniques such as structural hashing and resource-bounded BDD- and SAT-based analysis to identify gates which are functionally redundant across all states [13], as well as a variety of rewriting techniques such as [14] to reduce netlist size.

- **RET**: a min-area retiming engine, which reduces the number of registers by shifting them across combinational gates [10].
- **CUT**: a reparameterization engine, which replaces the fanin-side of a *cut* of the netlist graph with a trace-equivalent, yet simpler, piece of logic [15].
- **LOC**: a localization engine, which isolates a cut of the netlist local to the targets by replacing gates by primary inputs. **LOC** is an overapproximate transformation, and uses a SAT-based refinement scheme to prevent spurious counterexamples [15].
- **ISO**: a structural isomorphism detection engine, which equivalence-classes isomorphic targets, such that only one representative target per equivalence class needs to be solved by a child engine flow [16].
- **MOD**: a structural state-folding engine used to abstract certain clocking and latching schemes, generalizing the techniques presented in [17], [18].
- **SAT**: a hybrid-algorithm SAT solver based upon [13], which interleaves redundancy removal and structural rewriting with BDD- and SAT-based analysis.
- **RCH**: a BDD-based reachability engine.
- **IND**: a SAT-based induction engine which uses unique-state constraints.
- **BIG**: a structural target-enlargement engine, which replaces a target by the simplified characteristic function of the set of states which may hit that target within k time-steps [19].
- **SCH**: a semi-formal search engine, which interleaves random simulation (to identify *deep*, interesting states) and symbolic simulation (using either BDDs [20] or the **SAT** engine) to branch out from states explored during random simulation.
- **EQV**: a sequential redundancy removal engine based upon the algorithm described in Section IV-A [8].

Note that, rather than relying solely upon the flow of Figure 1 as our core SEC process (with a multi-algorithmic "prove" step), we have encapsulated this sequential redundancy elimination flow as the **EQV** engine. This enables us to apply an arbitrary sequence of engines *before* **EQV**, to exploit characteristics of the problem which may enable alternate algorithms such as **ISO** to more quickly reduce its domain – as well as *after* **EQV**, to leverage the sequential redundancy removal as a synergistic preprocessing to subsequent engines. This flexibility is often critical for leveraging **EQV** in property checking, wherein redundancy removal alone may be inadequate to solve the problem.

Encapsulating the sequential redundancy elimination flow as the **EQV** engine has non-obvious benefits even within the "prove" step. For example, on very large designs with millions of gates, the cost of computing an optimal set of equivalence classes may become prohibitive. It may thus be advantageous to arrive at that optimal grouping in phases. E.g., we may first instantiate an **EQV** that attempts only an accurate (yet incomplete) 1:1 register pairing. We may then leverage a more aggressive **EQV** instance on the suboptimal speculatively-reduced model, attempting to find 1:1 *gate* pairings. This may be followed by yet another **EQV** instance attempting to find general N:M gate pairings, possibly leveraging a variety of other transformations in this flow to further reduce the domain of the problem. For the largest designs, often the most efficient strategy is that of finding efficient transformations that safely chip away at size, until more exhaustive, albeit expensive, algorithms become applicable. In this example, we may defer the need for heavy-weight proof analysis until the most deeply-nested **EQV** instance when the problem domain is at its smallest, vs. suffer suboptimal merging or prohibitive proof resources directly in the first **EQV** instance.

One pronounced benefit we have noted lies in the use of **LOC** on the speculatively-reduced model. This is particularly useful when a redesigned portion of the design relies upon satisfiability don't-cares (SDCs) from its fanin cone to ensure equivalence, and when the sequential depth of logic in that cone precludes inductivity. By using a localization-refinement scheme to discharge the miters in the speculatively-reduced model, only the subset of the cone of influence necessary to ensure those SDCs will be included for proof analysis. In contrast, applying **LOC** without speculative merging is often highly inefficient for SEC problems, since it tends to degrade into including the entire cone of influence of the miter to ensure that no mismatches can occur, vs. including only the speculatively-merged subset of that cone needed to ensure the SDCs. Additionally, the localized cone may become quite large in cases, precluding an efficient proof. The cutpointing inherent in **LOC** opens up great reduction potential for transformations such as **RET** or **CUT**, possibly followed by additional **LOC** instances and other engines, until the size becomes amenable for **RCH** or optimal N:M gate pairing in **EQV**[15].

Due to the complexities of SEC for dissimilar designs, numerous dedicated SEC algorithms have been proposed for specific types of transformations. For example, in [21] a technique is proposed to derive a *retiming invariant* that may be inductively discharged more simply than could direct equivalence candidates. Such algorithms are somewhat complementary to ours, in that they could be incorporated as engines within our TBV framework, and our richer set of algorithms could in cases be used to discharge their resulting proof obligations. While such dedicated solutions are powerful and well-motivated, we have nonetheless found our flexible TBV approach to practically superset such dedicated techniques. For example, we have found that iterations between our **EQV** and **RET** engine tend to capture redundancies arising from gates that are equivalent modulo a time skew [22], even in cyclic designs where retiming and resynthesis are performed in conjunction with other transformations such as FSM re-encoding. This strategy also heuristically captures *k*-th invariants [23], which are redundancies that hold only after time *k*. The state re-encoding performed by engines such as **RET** and **MOD** is often effective in enhancing the inductiveness of properties of designs [24], as is **BIG** [25]. SEC of loop-free circuits may be efficiently reduced to CEC in our framework using techniques such as **RET**, **COM**, and **CUT**, or structural diameter bounding [19], without a need for dedicated algorithms to achieve scalability [26], [7].

For the most difficult problems, we have found that we may need dozens of nested engines to enable a solution. To automate the intricate problem of finding the best-tuned algorithm flow for a given problem, our system utilizes an *expert system* engine configured with a set of rules about commonly useful engine sequences, which orchestrates the scheduling of the engines while *learning* by trial and error about the effectiveness of the various engines on the problem at hand [27]. While for the largest and most complex problems there are still cases where an expert user can find a tuned configuration that eludes effective solution by the expert system engine, this engine has demonstrated to be much more powerful as a default TBV configuration than any pre-packaged set of engine sequences we have found. We estimate that the use of the expert system engine reduced the need for manual tuning of configurations by approximately an order of magnitude. Though like all aspects of SixthSense, the expert system engine continues to evolve in strength and capability.

V. Case Studies

Over the years, SixthSense has been used for many thousands of SEC runs, and has exposed far too many design flaws to count. Example SEC applications include:

- Verification that synthesis tuning, e.g., retiming or addition of sequential redundancy to enhance timing, does not alter functionality.
- Verification that a *technology remap* to a new latching and clocking schemes does not alter functionality.
- Verification that the modification of debug, test, and initialization logic does not alter functional behavior.
- Verification that a *backward-compatible* operational mode of a manually-redesigned component truly brings the redesigned component back to its prior behavior. Similarly, verification that a redesign affecting only specific types of input stimulus (e.g., a certain instruction type) preserves equivalence if those input stimuli are disallowed.
- Verification that clock gating functionality (which disables clocks to certain state elements when those elements are not required to update) does not alter design behavior during *care* time-frames.
- Verification that RTL implementations match the behavior of abstract reference models [28].

The design component sizes at which SixthSense is run typically range from several hundred state elements to well over 100,000. Table I illustrates the results of several sequential equivalence checks, where we detail the set of engines that solved the particular problems, the size of the resulting problem at various stages of the proof, and the total resources required to complete these equivalence proofs. Refer also to [8] for more examples.

The *VSU* is a vector / scalar unit, comprising a floating-point unit plus a vector arithmetic unit. The manually-performed design transformations were significant, including substantial redesigns of trace, debug, and ABIST (Array Built-In Self-Test) logic, timing optimizations, and a remapping to new latch library cells. After black-boxing certain memory arrays, which were independently verified by SEC, the earlier design comprised 82,151 registers, and the redesign comprised 82,165. Though this may appear to be a nearly 1:1 state element correlation, there were 17,062 registers that could not be paired across the two versions of the design even with sequential analysis.

The *IU* is a PowerPC instruction unit. Due to a hierarchy redesign that changed its interface, we performed this check at its parent hierarchy level, black-boxing unneeded child entities. The design changes in this unit were major, including a redesign of debug and trace logic that reduced an array and its datapath by one half, rearrangement of several design hierarchy points, and the addition of gating to certain clocks. After black boxing unchanged memory arrays, the earlier design comprised 132,081 state elements; the redesign comprised 109,493 state elements.

MUL is a 64-bit multiplier. Its redesign included a retiming of the pipeline structure, and a new algorithmic implementation of the *sign* processing of the carry-save adder tree. The pre-optimized design comprised 1,623 state elements; the optimized design comprised 2,145.

VI. Concluding Remarks

We have described our framework for sequential equivalence checking (SEC) at IBM. Our solution has been driven by a need for scalability and for applicability across arbitrary design transformations, including those that preserve neither initialization sequences nor even strict functional equivalence without the addition of *normalization logic* to compensate for the inequivalences. We have thus adopted a multi-algorithmic solution for performing SEC against designated initial states. Our framework has proven to be

VSU	Initial	COM	EQV[1]	RET	COM	MOD	IND	LOC[2]	EQV[3]	LOC	CUT	RCH
Inputs	2751	2427	2131	4271	3285	2402	2380	2348	2345	63	44	
ANDs	157359	693362	459973	433095	419657	284621	280200	275000	252750	441	529	
Registers	157359	125180	71121	63579	63575	32038	31546	31081	26390	94	94	5109 s
Targets	1666	1666	4852	4640	4640	2636	1211	615	1	1	1	2.5 GB

IU	Initial	COM	EQV[1]	IND	LOC[2]	COM	MOD	LOC[4]	CUT	EQV[3]		
Inputs	5548	3020	3018	1775	1775	1775	2905	605	370			
ANDs	1154650	565513	299243	206756	202935	198512	127826	2356	1905			
Registers	239898	141987	71788	50597	50404	50033	37022	593	582	2378 s		
Targets	2234	2234	2371	552	304	304	42	1	1	2.9 GB		

MUL	Initial	COM	EQV[5]	EQV[6]	EQV[3]							
Inputs	366	136	135	135								
ANDs	34878	23386	14561	12422								
Registers	3231	1935	1356	803	22068 s							
Targets	68	67	65	65	600 MB							

TABLE I: SEC Experiments. [1]Analysis of speculatively-reduced model shown; the proof of all targets on this model implies the proof of the initial targets. [2]This **LOC** instance is used to eliminate some non-inductive targets; though not shown, the configuration **RET,COM,CUT,RCH** was used to eliminate these targets. [3]This **EQV** performs full N:M gate merging. [4]This **LOC** performs a case split per target; the largest resulting localized cone is shown. [5]This **EQV** performs 1:1 register merging only. [6]This **EQV** attempts to prove constant gates only. Total resources (at 1.7GHz) across all targets shown in final column.

very powerful, scaling up to designs with 10,000s of state elements and beyond. Its efficiency in exposing intricate design flaws is causing SEC to become a standard part of design methodologies within IBM. While our system is powerful, SEC is a PSPACE-complete problem, hence we continuously are in the process of improving the capacity and automation of our solution through research and development. We strongly encourage the research community to continue working along these lines as well.

References

[1] A. Kuehlmann and C. van Eijk, *Combinational and Sequential Equivalence Checking*, in *Logic Synthesis and Verification*. Kluwer Academic Publishers, 2004.

[2] Z. Khasidashvili, M. Skaba, D. Kaiss, and Z. Hanna, “Theoretical framework for compositional sequential hardware equivalence verification in presence of design constraints,” in *ICCAD*, Nov. 2004.

[3] J. Ludden et al., “Functional verification of the POWER4 microprocessor and POWER4 multiprocessor systems,” *IBM Journal of Research and Development*, Jan. 2002.

[4] C. A. J. van Eijk, “Sequential equivalence checking without state space traversal,” in *DATE*, March 1998.

[5] C. Pixley, “A theory and implementation of sequential hardware equivalence,” in *TCAD*, Dec. 1992.

[6] V. Singhal, C. Pixley, A. Aziz, and R. Brayton, “Theory of safe replacements for sequential circuits,” in *TCAD*, Feb. 2001.

[7] Z. Khasidashvili and Z. Hanna, “SAT-based methods for sequential hardware equivalence verification without synchronization,” in *ICCAD*, 2003.

[8] H. Mony, J. Baumgartner, V. Paruthi, and R. Kanzelman, “Exploiting suspected redundancy without proving it,” in *DAC*, June 2005. *Extended version available.*

[9] V. Singhal, C. Pixley, R. L. Rudell, and R. K. Brayton, “The validity of retiming sequential circuits,” in *DAC*, 1995.

[10] A. Kuehlmann and J. Baumgartner, “Transformation-based verification using generalized retiming,” in *CAV*, July 2001.

[11] S.-Y. Huang, K.-T. Cheng, K.-C. Chen, C.-Y. Huang, and F. Brewer, “AQUILA: An equivalence checking system for large sequential designs,” *IEEE Trans. Computers*, vol. 49, no. 5, May 2000.

[12] P. Bjesse and K. Claessen, “SAT-based verification without state space traversal,” in *FMCAD*, November 2000.

[13] A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai, “Robust Boolean reasoning for equivalence checking and functional property verification,” *TCAD*, Dec. 2002.

[14] A. Mishchenko, S. Chatterjee, and R. Brayton, “DAG-aware AIG rewriting: A fresh look at combinational logic synthesis,” in *DAC*, July 2006.

[15] J. Baumgartner and H. Mony, “Maximal input reduction of sequential netlists via synergistic reparameterization and localization strategies,” in *CHARME*, Oct. 2005.

[16] G. S. Manku, R. Hojati, and R. K. Brayton, “Structural symmetry and model checking,” in *CAV*, July 1998.

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

[18] J. Baumgartner, A. Tripp, A. Aziz, V. Singhal, and F. Andersen, “An abstraction algorithm for the verification of generalized C-slow designs,” in *CAV*, July 2000.

[19] J. Baumgartner, A. Kuehlmann, and J. Abraham, “Property checking via structural analysis,” in *CAV*, July 2002.

[20] V. Paruthi, C. Jacobi, and K. Weber, “Efficient symbolic simulation via dynamic scheduling, don’t caring, and case splitting,” in *CHARME*, Oct. 2005.

[21] M. Mneimneh and K. Sakallah, “REVERSE: Efficient sequential verification for retiming,” in *IWLS*, 2003.

[22] S.-Y. Huang, K.-T. Cheng, and K.-C. Chen, “On verifying the correctness of retimed circuits,” in *GLSVLSI*, Mar. 1996.

[23] F. Lu and T. Cheng, “Sequential equivalence checking based on K-th invariants and circuit SAT solving,” in *HLDVT*, Dec. 2005.

[24] M. Wedler, D. Stoffel, and W. Kunz, “Exploiting state encoding for invariant generation in induction-based property checking,” in *ASP-DAC*, Jan. 2004.

[25] M. Awedh and F. Somenzi, “Increasing the robustness of bounded model checking by computing lower bounds on the reachable states,” in *FMCAD*, Nov. 2004.

[26] Z. Khasidashvili, J. Moondanos, and Z. Hanna, “TRANS: Efficient sequential verification of loop-free circuits,” in *HLDVT*, 2002.

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

[28] C. Jacobi, K. Weber, V. Paruthi, and J. Baumgartner, “Automatic formal verification of fused-multiply-add FPUs,” in *DATE*, March 2005.