The Art of Semi-Formal Bug Hunting

Pradeep Kumar Nalla
IBM Systems, Bangalore, India

Raj Kumar Gajavelly
IBM Systems, Bangalore, India

Jason Baumgartner
IBM Systems, Austin, TX

Hari Mony
IBM Systems, Austin, TX

Robert Kanzelman
IBM Systems, Rochester, MN

Alexander Ivrii
IBM Research, Haifa, Israel

ABSTRACT

Verification is a critical task in the development of correct computing systems. Simulation remains the predominantly used technique to identify design flaws, due to its scalability. However, simulation intrinsically suffers from low functional coverage, hence often fails to identify *all* design flaws. Formal verification (FV) is a promising approach to overcome the coverage limitations of simulation, due to its *exhaustiveness* – which enables it to identify intricate design flaws too complex to practically find using simulation. However, automated FV techniques have scalability drawbacks that limit the size of design components that can be formally verified. One of the key strengths of FV techniques is their use of symbolic reasoning, to efficiently explore a huge number of individual scenarios that would be intractable using simulation. When used in an incomplete manner, the scalability challenges of these algorithms are lessened, enabling efficient and relatively scalable *semi-formal bug hunting*. Nonetheless, to yield a robust industrial-strength solution, the individual components of such a system – many being heuristic – must be highly tuned, and integrated and orchestrated in an intricate manner. In this paper, we overview the various components useful in a scalable semi-formal search framework, introducing several novel powerful techniques and providing experimental data to illustrate the strengths, weaknesses, and complementary nature of the various techniques.

1. INTRODUCTION

Functional verification is the process of establishing the correct behavior of a computing system. This task is obviously an essential one: deploying a flawed system entails various risks, from the financial cost of repairing the system, to financial and legal damages caused by incorrect computations, or even to the loss of human life or environmental disaster. *Security verification* is a related concern, ensuring that a system is not unknowingly malicious or vulnerable to malicious control. The complexity of verification grows super-linearly with respect to the complexity of the system being verified, and is widely acknowledged as dominating contemporary semiconductor development costs. In a semiconductor setting, there are additional critical verification tasks including *equivalence checking*: verifying that two versions of a system behave equivalently – e.g., pre- and post-logic synthesis.

Due to its scalability, *simulation* remains the predominantly used technique to identify design flaws. However, simulation intrinsically suffers from low functional coverage, hence often fails to identify *all* design flaws. Simulation furthermore requires a significant manual investment to achieve a reasonable amount of coverage – e.g., development of coverage models and domain-specific testcase generators. Formal verification (FV) is a promising alternative, offering exhaustive analysis to not only expose the most intricate design flaws, but ultimately to yield proofs of correctness. However, automated FV techniques have scalability drawbacks that limit the size of components that can be exhaustively verified, hence FV often becomes a manually-intensive task itself. Nonetheless, FV has become as indispensable to the semiconductor industry as simulation, with key application niches such as equivalence checking for which it is the only practical solution. Contemporary verification methodology thus relies both upon simulation and FV techniques.

One of the key strengths of FV techniques is their use of symbolic reasoning to efficiently explore a huge number of individual scenarios that would be intractable using explicit approaches such as simulation. When used in an incomplete manner, the computational requirements of these algorithms are lessened, enabling efficient and relatively scalable *semi-formal search* for bug-hunting. Nonetheless, to yield a robust industrial-strength semi-formal solution, its individual components must be highly tuned for performance, integrated and orchestrated in an intricate and adaptive manner. While numerous components useful in a semi-formal bug hunting framework have been proposed in prior literature, there is relatively little work addressing the effective integration of these various techniques in a state-of-the-art solution. In this paper, we detail the various techniques essential to such a system, including several novel techniques and powerful extensions to existing methods. Experimental data demonstrates the utility of these techniques, and in cases – the lack thereof. We furthermore provide insight into the core reasons for the witnessed performance.

In cases, the criticality of functional correctness of a system is so great that it mandates a comprehensive purely-formal approach. The utility of a semi-formal framework may thus be questioned by purists. Regardless of the do-

ICCADÍ6, November 07-10, 2016, Austin, TX, USA
 ISBN 978-1-4503-4466-1/16/11…$15.00
DOI: http://dx.doi.org/10.1145/2966986.2967079

main, in our experience, such a framework is of utmost value, since: **(1)** if a verification task will ultimately fail, the sooner the fail is identified the better. Sometimes a proof is difficult to obtain solely because of an error. Discovering the flaw early will save effort on behalf of the verification team, and will benefit the design team by knowing of and being able to rectify the flaw as soon as possible. **(2)** Even given a bug-free system, there are many useful semi-formal methods to boost the scalability of a proof procedure. E.g., scalable approaches for identifying functionally redundant logic first identify inequivalences between pairs of gates using semi-formal techniques before attempting to prove equivalences: eliminating redundancy can significantly improve the scalability of verification, and is the cornerstone of equivalence checking. Invariant-learning techniques may benefit from efficient methods for eliminating invalid candidate invariants. Reachable state information can make IC3 more effective in its invariant generalization [10]. The proposed methods thus have value in bug-hunting frameworks, proof frameworks, and even synthesis frameworks.

2. PRELIMINARIES

We focus on the verification of hardware designs, which may be viewed as Finite State Machines (FSMs) derived in the standard way from synthesized netlists. Netlist constructs which define states include bit-level state variables such as latches, and memory primitives such as *RAM*. We assume a discrete-time logic view of a digital design, with combinational gates having 0 propagation delay from their inputs to outputs, and sequential gates taking next-state values at the beginning of every timestep.

Definition 1 A *FSM* M is a 6-tuple $A = (S, I, \delta, O, \lambda, S_0)$, where $S = s_1, ..., s_n$ is a finite set of states, I is a finite input alphabet, $\delta : S \times I \rightarrow S$ is the next-state function, O is a finite output alphabet, $\lambda : S \times I \rightarrow O$ is the output function, and $S_0 \subseteq S$ is the initial-state set.

Definition 2 A *trace* is an ordered sequence of (S, I, O) tuples beginning with an element of S_0, and with successive elements consistent with δ.

Definition 3 A *safety property* is a FSM output representing a verification objective: to obtain a *counterexample* trace illustrating the assertion of that output, or to prove that no such trace exists.

Definition 4 A *constraint* is a FSM output which restricts the set of legal traces: a trace is only valid if each constraint remains asserted for the finite duration of the trace.

Depending on the context, properties may represent not only primary verification objectives, but also secondary objectives such as candidate invariants of a netlist, or redundancy elimination opportunities. In a semi-formal search context, secondary properties referred to as *lighthouses* are often useful to help guide the search into rare netlist behaviors, or states deemed near to a property failure.

Constraints are useful netlist constructs to eliminate illegal input scenarios from a verification *testbench*. They also enable *assume-guarantee* based proof frameworks. However, constraints pose challenges to the efficiency of semi-formal search frameworks: simulated states that violate constraints must be discarded, and states that lead to a constraint violation may be fruitless in bug hunting. We refer to states that violate a constraint under all input valuations as *dead-end states*. For states that have at least one legal input valuation satisfying the constraints, the technique of [16] to synthesize logic natively satisfying such constraints is highly-effective at boosting semi-formal search scalability.

Simulation is a technique for producing a trace relative to a single sequence of valuations to inputs, often derived using a form of random generation [1]. Much literature exists on ways to accelerate simulation, e.g., **(1)** event-simulation techniques to avoid re-evaluating gates whose values have not changed from the previous timestep; **(2)** exploiting algorithmic parallelism which includes partitioning the netlist and evaluating different partitions on different host machines; and **(3)** exploiting data parallelism which evaluates different independent simulation runs on different host machines. We discuss our use of simulation in Section 8.

Bit-parallel simulation is a computationally efficient way to exploit data parallelism, concurrently deriving a set of traces using a single host machine instruction to atomically simulate the behavior of a gate relative to multiple vectors. Even with bit-parallel extensions, a significant drawback of simulation is that the fraction of netlist behavior that can explored explicitly is often very small.

Bounded model checking (BMC) is a technique to symbolically reason about the behavior of a netlist under all possible input valuations for a bounded number of timesteps, relative to a set of properties and constraints. BMC constitutes a powerful bug-hunting framework itself, given its relative scalability when using a state-of-the-art incremental SAT solver. Nonetheless, BMC does have capacity limitations, precluding it from exploring deep states of large netlists.

Both simulation and BMC are essential components of a robust semi-formal search framework. The purpose of this paper is to study effective techniques for falsifying properties that are too improbable to solve using random simulation alone, and too deep to expose using BMC alone. We furthermore strive for an industrial-strength solution which can scale to multi-million gate designs, rendering explicit-state model checking techniques inadequate.

If the verification objective includes liveness properties, practically we have found that a prior conversion to safety [3] is highly desirable from a scalability perspective. The reason is that establishing a state history record in bit-parallel simulation is a significant computational overhead, whereas otherwise the memory requirements of bit-parallel simulation can be reduced to proportional to netlist size regardless of search depth. For simplicity of exposition, we hereafter assume that any RAM are bit-blasted to latches. However, tailoring a semi-formal framework for native RAM support is highly desirable to improve scalability.

Our experiments were performed on a combination of benchmarks from [11], and proprietary benchmarks, which were too difficult to solve using an aggressive simplifying sequence of reduction and abstraction techniques (including combinational constraint synthesis [16]) followed by 1 hour of BMC and 1M simulation vectors. These resulting simplified netlists are used in our semi-formal search experiments, allowing the reductions to improve the scalability of semi-formal search, as shadows the often most-effective strategy for solving the hardest verification problems. All experi-

```
while (!solved && trial < max_trials)
  iteration = 0;
  set latches to initial states;
  while (!solved && iteration < max_iters)
    perform BMC;
    perform simulation;
    update coverage;
    extract lighthouses and patterns;
    choose and set states for next iteration;
    iteration++;
  trial++;
```

Figure 1: Generic Semi-Formal Search Framework

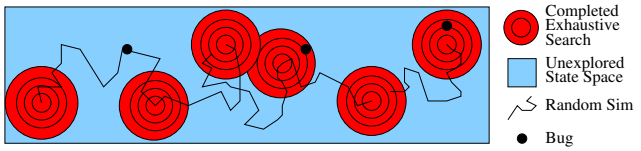

Figure 2: Semi-Formal State Search

ments were performed on a 64-bit Linux machine, using a single of 32 CPUs at 2.66 GHz with 32 GB main memory.

3. SEMI-FORMAL FRAMEWORK

Figure 1 illustrates a generic semi-formal search framework, leveraging BMC and simulation under configurable resource limits, beginning from specifically chosen states. This search is depicted in Figure 2, illustrating how iterations beginning from different starting states have the opportunity to explore a large sampling of the reachable state space, and ideally will seed BMC near to a property failure.

An often-effective strategy for semi-formal search orchestrates against *trials* and *iterations*. Each trial begins from the original initial states of the netlist. Each iteration begins from a set of states selected from prior search. The number of distinct states seeded per iteration is a nontrivial heuristic to balance: the more states chosen per iteration, the fewer distinct bit-parallel simulations that begin from each state, and the less scalable that BMC will generally be.[1] However, selecting more than one starting state per iteration can often yield greater coverage, since the degradation to bit-parallel simulation and BMC coverage is often sub-linear.

It is advantageous to perform multiple searches from the initial states, because **(1)** a netlist generally has multiple initial states, and property failures may only be reachable along a subset of these. **(2)** Generally, decisions made earlier in a trace may preclude the reachability of states later in that trace. **(3)** It is often desirable that counterexamples be as short as possible, since the manual effort involved in triaging counterexamples is often proportional to their length. Additionally, computational requirements of counterexample generation are proportional to trace length.

It is advantageous to perform multiple iterations per trial, because **(1)** BMC requires exponentially growing resources with search depth, so becomes prohibitive after a given depth. By seeding later iterations into deeper states, the astronomical coverage of BMC is leveraged from states that practically could not be reached using symbolic methods alone. **(2)** While simulation is not similarly scalability-challenged, it is advantageous to guide simulation to prevent it from wasting resources exploring uninteresting states. A simple motivating example is a netlist with a *reset* input: the probability of exploring a trace of length k without resetting the netlist (entailing redundant analysis) is 2^{-k}, as noted in [5].

The processes of measuring coverage, selecting states for later search iterations, and generating lighthouses and input patterns will be detailed in later sections. These techniques must be highly-tuned and instrumented in a careful manner to ensure that their computational overhead does not merely degrade the search coverage achieved.

[1]When constructing a BMC instance from multiple initial states, the most effective modeling we have found uses parametric input variables to select among the starting states, and constants for state variables in each starting state. This allows states with small hamming distance to map to constant state variables, reducing the size of the BMC problem and thereby increasing the scalability of BMC.

4. RARITY-GUIDED SEARCH

Rarity-guided simulation was proposed in [9], as a metric by which to rank states to be used for later search iterations. This technique proposes to partition the netlist into clusters of state variables, and measure how often states (projected onto these partitions) have been encountered during simulation. For a partition size of k and a netlist with N state variables, this is implemented using at least $2^k \times \lceil \frac{N}{k} \rceil$ counters, as partitions may generally be overlapping groups.

This approach is admittedly lacking in that: **(1)** reaching a new state does not guarantee that any partition will have a new sub-state identified; **(2)** a rare state does not imply nearness to a property failure. However, in industrial settings, the netlist is often far too large for explicit enumeration of most states anyway. A newly reached partition state *does* imply a newly reached concrete state, satisfying the goal of ensuring a better sampling of the state space across iterations compared to unguided search.

The utility of rarity-guided simulation was noted in [5], in the context of redundant gate identification. It was observed that unguided random simulation would quickly saturate in ability to expose incorrect candidate redundancies, leaving many to be falsified using heavier-weight proof techniques. Rarity guidance offers a scalable heuristic approach to increase the coverage of simulation, helpful to falsify many more incorrect redundancies. The approach of [5] selects the rarest states only *within the same* bit-parallel simulation timestep of prior iterations. We have found it significantly superior to select the rarest states *from arbitrary points* in the prior search history. This benefit is significantly greater in testcases with dead-end constraints, since the percentage of simulation vectors which do not violate constraints may be very small, hence limiting selection granularity becomes a severe restriction.

A partition size of 8 state variables was found effective in [5], and also in our experience, since this coverage model is small enough to be efficiently computed and offers useful concrete state differentiation. We have found that using even smaller partitions does not significantly improve scalability, though does significantly weaken rarity guidance since the small partitioned state cubes saturate too quickly. With careful tuning, this approach scales with little runtime overhead up to 16-latch partitions, albeit with a significant memory overhead. However, the utility of such larger partitions becomes questionable: the coverage space is so large ($2^{16} \times \lceil \frac{N}{16} \rceil$ vs. $2^8 \times \lceil \frac{N}{8} \rceil$) that it takes too long to converge in coverage, and using the uncovered partition states as lighthouses creates too much overhead.

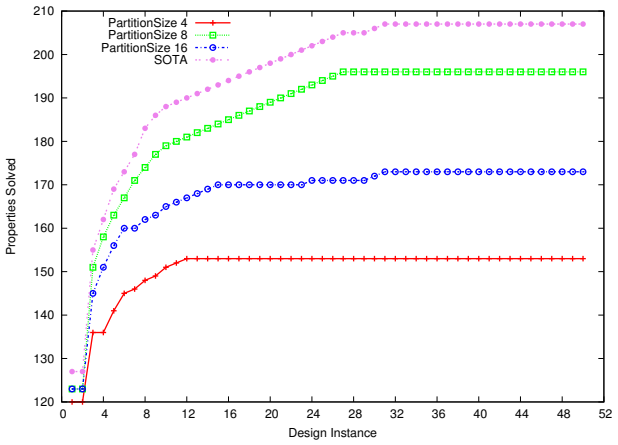

Figure 3: Sim Solves vs. Partition Size

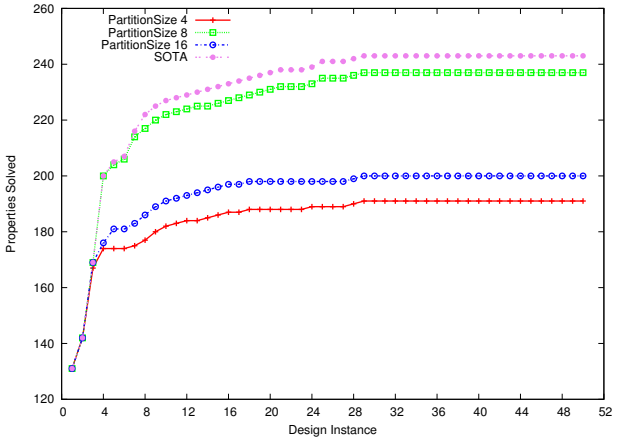

Figure 4: Sim + BMC Solves vs. Partition Size

Figure 3 vs. Figure 4 provides a set of experiments illustrating the ability to falsify properties using purely guided simulation vs. the combination of guided simulation and 180 seconds of BMC per iteration, respectively, for partition size 4, 8, and 16, and "SOTA" reflecting running all approaches concurrently on different host machines (a *State-Of-The-Art* portfolio approach). These experiments use fifty 64-bit words of bit-parallel simulation, with each iteration selecting fifty of the rarest states encountered from prior search: one per 64-bit word. In Figure 3 we used 20 simulation timesteps per iteration, with 100 iterations per trial. In Figure 4 we used 1000 simulation timesteps and BMC per iteration, with 10 iterations per trial. The number of trials per benchmark is identical in the various modes, though varies from 10 to 1000 depending on benchmark size to contain overall runtime.

These plots clearly show 8 latch partitions as the winning strategy, with relatively little cumulative benefit (exploiting unique solves) for the portfolio approach. Practically, we have found 16 latch partitions to be valuable to alternate to *after* saturating 8 latch partitions formed using different strategies, to offer some guidance into unexplored states once the most fruitful ones have otherwise been exhausted.

Fig. 5 depicts the number of bit-parallel simulation timesteps necessary to converge rarity coverage using different partition sizes, i.e. at which no additional counter updates occur within an iteration. We use a maximum count value of

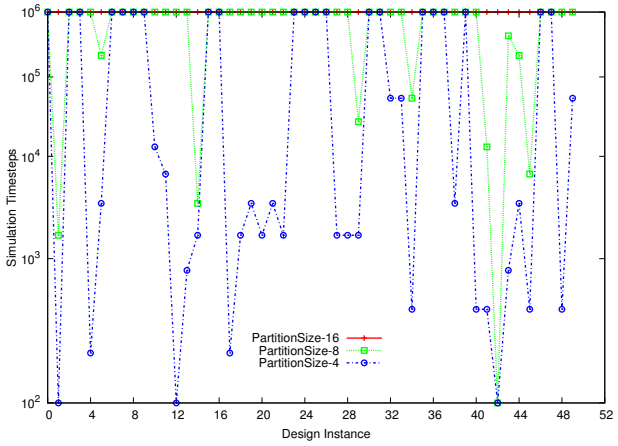

Figure 5: Simulated Timesteps until Saturation

65536, with 1000 simulation timesteps per iteration of fifty 64-bit words. This saturation criterion is useful for several purposes, including adapting search depth to the inherent depth of the netlist. Note that partition size 4 saturates quickly in many cases, which is its major shortcoming. In contrast, partition size 16 almost never saturates before 1M bit-parallel simulation steps, and even 8 often requires more than 1M simulation steps until convergence. On average, after each trial 45.78% counters saturated for partition size 4, 13.97% for partition size 8, and 0.03% for partition size 16, before convergence.

Overall, a weakness of 16 latch partitions is that they differentiate many more states than is practically useful, as will be discussed in Section 5.1. For this reason, it may seem desirable to assess saturation as too small of a *percentage* of counters updating. Though even with a very small percentage such as 0.1% of counters updating per iteration, this metric may misleadingly appear to saturate too early, in cases causing the much larger coverage space of 16 latch partitions to saturate *before* 4 or 8 latch partitions on the same benchmark.

4.1 Partitioning Strategy

The next major criterion of rarity-guided simulation is how to partition the netlist. In [9], it is proposed to group latches which significant *affinity*, overlapping at least 25% in next-state function support. The intuition is that this will group highly-correlated state variables, e.g., implementing the same state machine component. An improvement that we have found effective is to always group the highest-affinity pair of state elements first, in case there are too many high-enough affinity latches to fit within a single partition. Affinity-based partitioning will also leave a subset of latches unassigned: 10% of latches in these benchmarks were below 25% affinity to any others. We thus have found it helpful to use alternate criterion to group low-affinity latches. With careful tuning, affinity-based partitioning is fairly scalable even on large netlists, despite the inherent quadratic nature of comparing the support of pairs of state variables. Though its runtime is nontrivial: on average, affinity analysis consumed 9% of overall runtime of the following experiments, ranging from negligible to almost 50% on a 2.4M gate design.

Practically, we have found that the order obtained from a depth-first search (DFS) backward from the properties is usually comparable in quality and with negligible runtime.

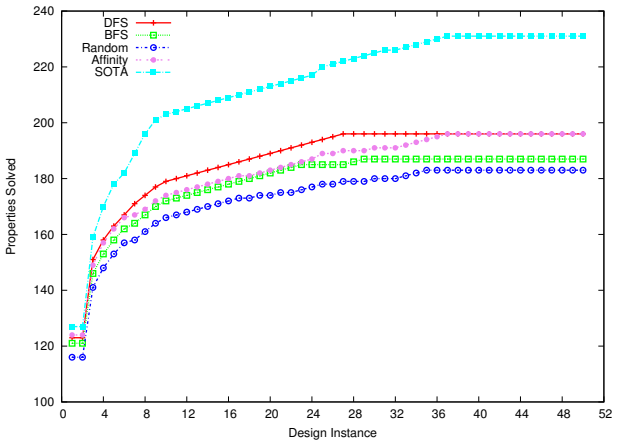

Figure 6: Sim Solves vs. Partitioning Type

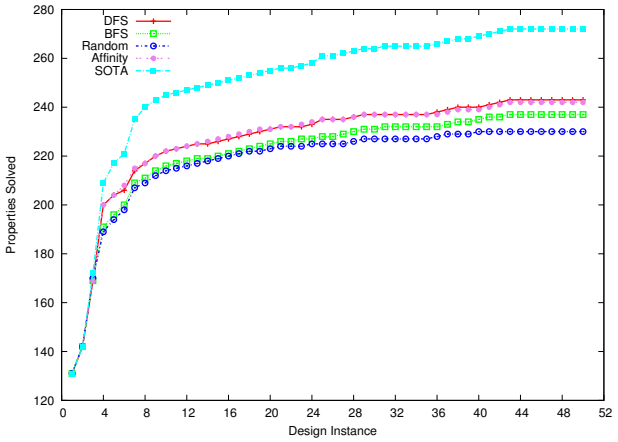

Figure 7: Sim + BMC Solves vs. Partitioning Type

We also use DFS ordering to group low-affinity latches. We contrast with breadth-first search (BFS) and random partitioning in Figure 6 and 7 for simulation alone vs. simulation and BMC, respectively.

These figures show that all techniques perform fairly well, with DFS and affinity-based being the consistent winners offering 7% more solves than random partitioning. A portfolio strategy of running each approach concurrently offers a significant improvement in total properties solved. This justifies one of our findings: when coverage saturates and lighthouse benefits are exhausted, it is beneficial to re-partition the netlist using an alternate strategy. It is further valuable to introduce randomness into every partitioning strategy, to break ties, thereby offering cumulative re-partitioning benefit across a large number of trials and iterations. To reduce the random noise due to this observation, hereafter all experiments are done using affinity-based partitioning.

One may be tempted to use even more precise state coverage in semi-formal search, e.g., state-hashing data structures common in explicit-state model checking [8]. However, our experience is that this is counter-productive. Aside from the computational overhead of state hashing, the goal of explicitly analyzing (with simulation) most reachable states is highly unrealistic for large netlists. BMC relative to seeded states offers much greater state coverage, albeit in a manner that cannot be enumerated. Furthermore, gaps in the coverage of state-hashing records do not lend themselves to useful lighthouses: their support is too large, and their number is too great to be meaningfully used as sub-properties (similar to our observation with 16-latch partitions). Instead, it is more fruitful to leverage rarity-guided simulation to focus the huge coverage of BMC from interesting deep states.

5. LIGHTHOUSE GENERATION

In addition to the original properties being falsified, it is often advantageous to generate secondary verification objectives to guide the semi-formal search. When these secondary objectives are encountered, the resulting state is qualitatively different from the prior search, hence is often useful to guide later search as a seeded iteration state or an input pattern (refer to Section 6). This can often increase the probability of encountering other sub-properties [12], and ideally be near enough to a failure of an original property to enable BMC to expose it.

We have found numerous criteria as useful lighthouses: **(1)** uncovered state partition values that were never (or rarely) encountered during semi-formal search [9, 5, 12]; **(2)** toggle-activity based lighthouses, i.e. gates that never or only rarely toggled during the search [9]; **(3)** exclusive-ORs or implications thereof between sets of gates that are equivalent/antivalent in all encountered states [13, 5]; **(4)** candidate invariants which may hold over small-support subset of the netlist (e.g., cut-based invariants or implication invariants [6]); or **(5)** abstract preimages of an original property (refer to Section 7). Approach **(3)** was borrowed from equivalent-gate identification frameworks [13, 5], though we have found it to be a powerful bug-hunting framework in itself due to its scalability in exposing rare behaviors of relevance to the netlist. Approach **(4)** extends this reasoning to groups of more than 2 gates, while still containing the number of lighthouses to linear with respect to netlist size.

5.1 Exploring Relevant Behaviors vs. Arbitrary New States

The work of of [9] proposes that low-activity lighthouses apply solely to state variables. In our experience, *all* gates should be included in activity-based analysis, as this offers exponentially more granular of state resolution with very little overhead. Furthermore, gates inherently reflect the manner in which the netlist reacts to symmetry groups of states. This observation prompts a reflection on the nature of semi-formal search.

Recall that the focus of this work is on techniques for effective bug-hunting in netlists too large for direct application of BMC or explicit-state model checking. The fraction of reachable states which can be explicitly explored in such netlists is vanishingly small. It is uncommon that a netlist reacts *qualitatively* differently to *every* concrete state. For example, a netlist with a counter could transition to a unique next-state from each current-state, though there are likely a smaller number of state symmetry groups wherein the netlist behaves qualitatively differently – e.g., where the counter saturates or overflows or resets, vs. performs "yet another increment." If the netlist reacts to a specific count value, it will likely have a gate which (de)asserts only at that count value. Perhaps the counter is too large to be contained in a single rarity partition. Thus, by including gates as candidate lighthouses, we can readily discern such netlist-relevant symmetry groups which otherwise would be impractical to capture as "different rare states."

Clearly, it is more essential to explore at least one state

per symmetry group relevant to the netlist, vs. many states within the same symmetry group, to achieve highest coverage. This is why redundancy removal frameworks are often useful to find intricate design flaws [13, 5]. This is also why empirically it is more useful to form rarity partitions with high-affinity latches vs. arbitrary partitioning: the probability that different partition states affect the netlist in a qualitatively different way, vs. "merely reflect a different state," is much higher. However, a focus on toggling every gate is not an adequate strategy in practice: even the properties are simple gates in the synthesized netlist. Barring a scalable enough proof system to prove the untoggled gates as constants – without a more meaningful notion of state coverage, there is no indication of whether the properties are likely unfalsifiable or whether the netlist is just too complicated to sufficiently explore using semi-formal analysis.[2] Guided simulation is thus a well-motivated approach to better ensure the exploration of states representing qualitatively different behaviors of the netlist first, before degrading to merely searching from arbitrary different states. Our findings of the limited utility of large-partition state coverage justifies this observation.

The experimental value of lighthouse generation will be discussed in the following section.

6. PATTERN-GUIDED SEARCH

In addition to using *states* which hit lighthouses as starting points for later search iterations, it is also valuable to learn *input sequences* along which the lighthouses were hit (particularly via BMC) to guide later search. Granted, the impact of replaying an input sequence will vary depending on the starting state from which it is applied. Though applying an input sequence to simulation will guide it through a specific sequence of states which may be too improbable to expose otherwise. In fact, sometimes an input sequence obtained from a lighthouse may be the key to exposing an intricate bug when applied from an alternate state. Input sequences should generally be minimally-assigned, offering greater coverage through randomization of unassigned inputs, and greater *compatibility* for concurrent application of multiple sequences with disjoint support, or with common inputs assigned to common values. Such input patterns may also be used to constrain BMC, as a form of concolic testing combining symbolic with concrete values. Though practically, we have had little success on large netlists with constraining BMC vs. combining BMC with guided simulation.

As the number of lighthouses may be very large, leveraging them through pattern reuse vs. seeding as states into semi-formal iterations has multiple benefits. **(1)** Similar to resimulation of BMC traces in redundancy identification frameworks [13, 5], the randomization of unassigned input patterns often yields additional rare scenarios beyond the lighthouse encountered with BMC, when applied to the same starting state. **(2)** Replaying the BMC scenario in simulation allows existing state prioritization to determine *which* scenarios to seed into later iterations, useful if there are multiple lighthouses hit (nearly) simultaneously. **(3)** Existing state prioritization can further determine *where* along a lighthouse-hitting trace to seed a later iteration. In cases,

[2]The latter can be particularly challenging in a netlist with dead-end states, precluding the ability to randomly simulate a reasonable sampling of the reachable states.

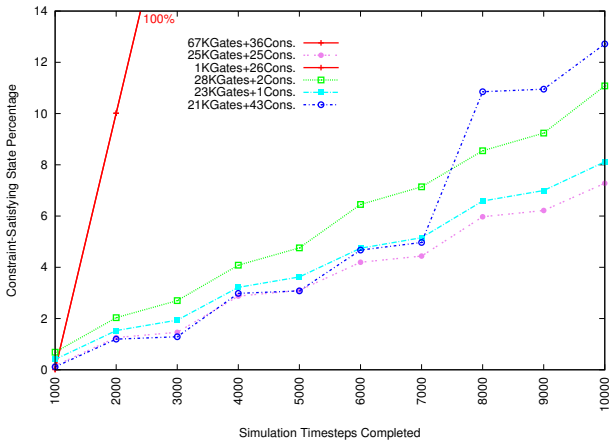

Figure 8: Constraint-Satisfying Patterns

the optimal choice may actually be earlier than the hit; seeding later may bypass too many scenarios of interest.

The benefit of pattern reuse is particularly pronounced in a netlist with dead-end constraints, as the fraction of randomly-generated inputs which satisfy the constraints may be very small. It is furthermore advantageous to directly generate input sequences that satisfy constraints for some bounded window to guide random simulation, as noted in [2], to avoid simulating a dead-end state. However, the overhead of computing constraint-satisfying inputs for every bit-parallel pattern is too great in practice. Instead, we have found the following extensions to be useful: **(1)** Only use constraint-satisfying patterns for a subset of parallel simulations, enabling this technique only if the percentage of wasted simulation patterns becomes significant. **(2)** Instead of generating constraint-satisfying patterns from every current simulation state, to generate patterns only from a subset of states and reuse them from other states. While this may fail to avoid dead-end states from incompatible states, it entails much less computational overhead.

Figure 8 illustrates the importance of constraint satisfying patterns in dead-end state avoidance, using a sequence of 1000 bit-parallel simulation timesteps per iteration. These are all examples where oblivious random simulation encountered dead-end states within a few timesteps. To enable simulation to reach deeper states, we add an extra property to BMC (for every BMC timestep checked) to generate constraint-satisfying input patterns for a single bit-parallel simulation state at the beginning of each iteration. We reuse each of these resulting input sequences for a percentage of all starting states for the iteration. On average, 16% of the 50×64 simulation vectors per iteration were directed with these input patterns. The y-axis indicates what percentage of the simulation patterns avoided dead-end states. The percentage of constraint-satisfying patterns grows across iterations due to seeding constraint-satisfying rare states into these later iterations, similar to the setup of Figure 3. Effectively, rarity-guided search benefits from learning diverse constraint-satisfying *states*. Note that one of the examples achieves 100% dead-end state avoidance after two iterations, as all of its selected rare states preclude dead-end states.

Additional value of lighthouse-based pattern reuse will be discussed in Figure 9.

7. ABSTRACTION-GUIDED SEARCH

Much recent research in semi-formal search has focused upon abstraction guidance [4, 15, 7, 14], computing a sequence of abstract preimages of a property to use as an underapproximate distance measure between a concrete state and a failure. The proposed approaches generate the abstraction either manually [15, 7], or automatically using localization – replacing some internal logic by primary input variables [4, 14]. The objective of abstraction-guided search is to iteratively tunnel to shallower abstract preimage state sets, using simulation [7], BMC [4], or both [15].

There are two fundamental challenges in abstraction-guided search: **(1)** States which are abstract-distance-1 from each other may be arbitrarily far apart in the concrete netlist. Though ideally, there will be significant temporal correlation between abstract and concrete states: e.g., a concrete counterexample may need to pass through iteratively shallower abstract preimage states, even if the concrete counterexample is much longer than the abstract sequence. **(2)** The coarseness of abstract vs. concrete distance often undermines the desired temporal correlation between the two: there are often many *misguided states* in the abstract preimages whose successors are all in equi-distant or even in further abstract preimages.[3] Practically, it is difficult to distinguish between the two cases during semi-formal search and decide when to truncate deeper analysis of a preimage-based state. A poor abstraction will suffer from too many misguided states to offer valuable guidance. Prior work has addressed this challenge by either refining to overcome the coarseness of an existing abstraction [4, 14], or by tailoring the search process to avoid *dead-end* and *misguided* states as they are explored and determined to be fruitless.

We use localization to automatically generate the abstraction, with no manual guidance whatsoever – as is often necessary for practical adoption in an industrial setting. BDD-based reachability provides the abstract preimages, used as *lighthouses* to guide the search. Lighthouses representing shallower abstract preimages are given higher priority than deeper ones when encountered in semi-formal search.

As noted in [7], we also find that while this technique can be very powerful, it is unreliable, and generally less useful on large netlists where an accurate-enough abstraction will be too large for preimage computation regardless of how the abstraction is obtained. Of the 50 benchmarks studied in detail, abstraction-guidance was able to solve 124 properties on 23, with 14 unique solves on two benchmarks. While it is straight-forward to limit the size of the abstraction even on large netlists, the core problem often arising is that the temporal depth of an abstraction small enough for preimage computation is too shallow, resulting in little opportunity for useful temporal guidance and, in reality, an overall degraded search due to misguidance. For the successful benchmarks, the maximum preimage depth was 1025 and average maximum depth was 99. For the unsuccessful ones, the maximum depth was 36 and average maximum was 2. This observation offers insight into whether it is worth attempting abstraction-guidance on a netlist whatsoever: if the maximum preimage depth of the best achievable abstraction is shallow, the likelihood of degraded search is higher.

[3]Prior work often referred to *misguided* states as *dead-end* states. We prefer the term *misguided*, as such states are not necessarily incapable of leading to failures along longer paths not adhering to abstract distance guidance. We reserve *dead-end* to refer to states which can *never* transition to a property failure, possibly due to violated constraints.

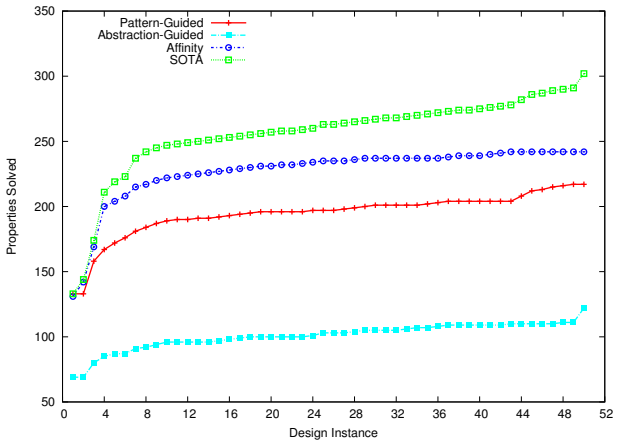

Figure 9: Sim + BMC Solves vs. Search Strategy

Figure 9 demonstrates the benefit of abstraction-guided search and lighthouse-based pattern reuse across a benchmark suite. It is noteworthy that rarity-guided search of coordinated bit-parallel simulation and BMC tends to significantly outperform any other individual technique presented in this paper for large netlists. It is thus used as a baseline comparison. However, there is significant cumulative portfolio value in all of the techniques discussed in this paper; no single technique subsumes the others across all benchmarks, as illustrated by the *SOTA* portfolio solving 20% more properties than rarity-guided search alone.

Overall, abstraction-guidance is an essential component of a robust semi-formal search portfolio. It can complement and outperform the other presented techniques in cases. Though, its utility degrades significantly as netlist size increases. It is worth noting the synergy between abstraction-guided search and the other techniques presented in this paper: rarity analysis, lighthouse- and pattern-guided search, and highly-tuned bit-parallel simulation all offer value in conjunction with abstraction-guided search, to yield a diverse sampling of rare states to explore when attempting to tunnel to shallower abstract preimages.

8. HIGH-THROUGHPUT SIMULATION

As bit-parallel simulation is a core element of semi-formal search, it is essential to tune its performance for high throughput. We have found the following techniques to be effective.

First: bit-parallel simulation is an effective way to exploit data parallelism: there is no practical overhead to simulating the netlist for 64 bit-level patterns (one *word*) vs. a single bit-level pattern on a 64-bit host machine. Practically, a much higher throughput may be achieved using multiple words: 50 were used in in [5]. We have found the optimal width to be significantly higher, though varies depending on the netlist being simulated as well as the host machine. We thus propose *adaptive bit-parallel simulation width*, evaluating throughput for a several-second interval across a span of widths to find the optimal. Our most effective semi-formal solution slightly biases the chosen width narrowly (to minimize memory requirements and simulation runtime per iteration), and uses binary search for optimal width within the best boundaries obtained from interval probing.

We illustrate throughput vs. width for a suite of netlists in Figure 10. Optimal throughput is achieved between 58 and

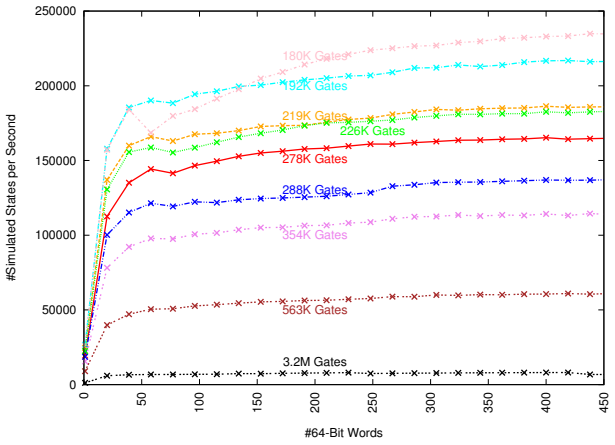

Figure 10: Simulation Throughput vs. Width

1293 words: imposing a 1% penalty per word, this decreases to a range of 39 to 172. Note the steep improvement in throughput until around 50 words, which plateaus and, if continued, eventually gradually decreases. An average of 20.4% higher throughput is achieved for optimal width vs. 50 words with average optimal width of 593.3, and 909.5% higher for optimal vs. a single word – which is 5820.8% higher than scalar simulation of a single pattern. The largest netlist of 3.2M gates has a throughput of 46.1 states per second for a single word, 483.1 at 50 words, plateauing at 1015.3 for 190 words.

Second: to minimize memory requirements, the simulator can be tuned to require only a single timestep of bit-parallel simulation data. This is achieved by performing all desired coverage analysis for a completed bit-parallel simulation timestep before moving to the next timestep, then mapping next-state functions to current-state values before simulating the combinational gates. This approach requires recording random number generation seeds so that when it is necessary to produce a temporal trace, the desired pattern can be recomputed independently of simulation width.

Third: for better memory locality and to minimize cache misses, it is most efficient to simulate the gates in an order that reflects netlist topology. The ordering of And/Inverter Graph node indices is often sufficient.

Finally, we have not found practical value in event simulation speedups to minimally propagate gate value changes. This is because in bit-parallel simulation, especially under rarity-guidance which maximizes diverse activity throughout the netlist, the chance that none of the words for a given gate requires an update is very small. Event simulation techniques impose significant overhead which degrades performance if even 1-2.5% of gates toggle [1]. We also have not found compelling value of partitioned netlist simulation, because rarity analysis is done for the entire netlist every timestep anyway. Nor have we found value in compiled-code simulation (translating the netlist itself into machine instructions), since host machines offer significantly better data-cache vs. instruction-cache performance. Instead, we have found it most efficient to optimize a basic bit-parallel netlist simulator, evaluating every gate at every timestep: a so-called oblivious evaluation strategy [1].

9. CONCLUSION

In this paper we presented a suite of techniques effective for semi-formal bug-hunting on large industrial netlists which are too complex for random simulation, BMC, or explicit-state model checking alone. We overviewed a variety of techniques proposed in prior literature, along with experimental evidence of their utility (or lack thereof), and offered insights into the reasons for this performance. We additionally described novel approaches to increase the utility of rarity-guided, lighthouse-guided, and pattern-guided search, as well as a novel adaptive bit-parallel simulation approach. The proposed techniques offer particular value in testbenches with dead-end constraints, to learn constraint-satisfying rare states and input sequences. Overall, a rarity-guided combination of BMC and bit-parallel simulation is consistently the winning strategy. Though each technique offers significant value on different benchmarks, with notable synergies between the techniques, contributing to a robust semi-formal search portfolio.

10. REFERENCES

[1] M. Bailey, J. Briner, and R. Chamberlain. Parallel logic simulation of VLSI systems. *ACM Computing Surveys*, 26(3), September 1994.

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

[3] A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. In *FMICS*, 2002.

[4] P. Bjesse, J. Kukula, and S. Inc. Using counter example guided abstraction refinement to find complex bugs. In *DATE*, 2004.

[5] R. Brayton, N. Een, and A. Mischenko. Using speculation for sequential equivalence checking. In *IWLS*, 2012.

[6] M. Case, A. Mishchenko, R. Brayton, J. Baumgartner, and H. Mony. Invariant-strengthened elimination of dependent state elements. In *FMCAD*, Nov. 2008.

[7] F. M. De Paula and A. J. Hu. An effective guidance strategy for abstraction-guided simulation. In *DAC*, 2007.

[8] P. Dillinger and P. Manolios. Fast and accurate bitstate verification for SPIN. *SPIN Workshop*, 2004.

[9] M. K. Ganai and A. Aziz. Rarity based guided state space search. In *GLSVLSI*, 2001.

[10] A. Gurfinkel and A. Ivrii. Pushing to the top. In *FMCAD*, 2015.

[11] Hardware Model Checking Competition 2015. http://fmv.jku.at/hwmcc15/.

[12] P.-H. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor, and J. Long. Smart simulation using collaborative formal and simulation engines. In *ICCAD*, Nov. 2000.

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

[14] K. Nanshi and F. Somenzi. Guiding simulation with increasingly refined abstract traces. In *DAC*, 2006.

[15] S. Shyam and V. Bertacco. Distance-guided hybrid verification with GUIDO. In *DATE*, 2006.

[16] J. Yuan, K. Albin, A. Aziz, and C. Pixley. Constraint synthesis for environment modeling in functional verification. In *DAC*, June 2003.