Approximate Reachability With Combined Symbolic And Ternary Simulation

Michael Case Jason Baumgartner Hari Mony Robert Kanzelman
IBM Systems and Technology Group

***Abstract*— Logic synthesis and formal verification both rely on scalable reachable state characterization for numerous purposes. One popular technique is over-approximate reachability analysis using an iterative ternary simulation. This method trades precision of reachability characterization for a high degree of computational efficiency. Although effective on many industrial designs, it breaks down when the design has registers that have complex initial states or has extremely deep deterministic subcircuits. In this paper, we improve upon the precision of ternary simulation-based approximate reachability while retaining its scalability by representing certain variables as symbols vs. *unknowns*, and by selectively saturating subcircuits which would otherwise preclude convergence. These techniques are particularly beneficial for enhancing the scalability of industrial sequential equivalence checking problems, occasionally solving such problems outright with no need for more costly and precise analysis.**

I. Introduction

Reachability analysis has many applications in contemporary verification and synthesis tools. For example, a design may be optimized using information about gates which are redundant in the reachable states; behavioral characteristics such as *oscillators* and *transients* may be identified and exploited for specific abstraction strategies; and properties may be solved using reachability information.

Unfortunately, exact reachability analysis is often computationally impractical, even for moderately sized designs. Approximate reachability analysis is thus often necessary, trading the precision of reachable state characterization for computational efficiency. Even when precise reachability analysis is ultimately necessary, it is often computationally beneficial to first apply faster approximate techniques to reduce the design before exact reachability analysis is performed.

One may perform approximate reachability analysis with ternary simulation [1] by letting signals take values in $\{0, 1, X\}$ as follows. Primary inputs are assigned X, and registers are assigned their initial values. Ternary simulation is then used to derive the next state. Computation proceeds in this way until a repetition of state values has been witnessed, indicating that an over-approximate reachability analysis has converged.

Reachability analysis with ternary simulation requires little runtime, often executing in seconds even on the largest industrial designs. Its reachability approximation is coarse but is precise enough to identify common artifacts in industrial verification and synthesis frameworks: inputs and registers that are constant due to testbench assumptions, simple internal equivalences, oscillating clocks, and transient signals. For this reason, ternary simulation-based reachability analysis is implemented in many logic synthesis and verification systems.

One key weakness of reachability analysis with ternary simulation is its inability to precisely characterize designs with complex initial values. Any registers with non-deterministic initial values are assigned X in the first iteration of reachability analysis, and because of the conservative nature of ternary simulation this X propagates to *all* fanout logic. For designs with non-deterministic initialization, this often precludes *any* useful reachable state characterization with this analysis.

A secondary weakness of reachability analysis with ternary simulation is that it may require an infeasible number of simulation steps to converge; designs containing large counters or "linear feedback shift register" type logic are often particularly problematic. This lack of convergence precludes any reachable state approximation.

In this paper we improve the precision and conclusiveness of reachability analysis with ternary simulation in two ways:

1) We utilize a symbolic representation and use this to represent initial states. We define reachability analysis over both ternary simulation and symbolic simulation. In practice, this adds little to the total runtime of approximate reachability and it improves the resolution substantially, particularly for industrial *Sequential Equivalence Checking (SEC)* problems.
2) We introduce a *saturation* technique to enable convergence without loss of useful reachable state characterization.
3) We additionally introduce extensions to ternary simulation-based application domains of redundancy removal, phase abstraction, and transient elimination to generalize them accordingly given our symbolic techniques.

Section II describes the related work in this field. In Section III we provide preliminaries, including an overview of reachability approximation using ternary simulation. Section IV describes symbolic simulation, and Sections V and VI incorporate symbolic simulation into approximate reachability. Section VII introduces our saturation technique which helps convergence in cases. Finally, experimental results are given in Section VIII.

II. Related Work

A significant amount of work exists in the field of approximate reachability characterization. Due to space limitations, we focus only upon those which rely upon ternary simulation vs. more expensive and precise techniques.

The use of reachable state characterization via ternary simulation and its applications within general model checking

was proposed in [1]. They note that this technique can identify a useful subset of redundant gates, whose simplification greatly enhances the scalability of subsequent verification. They also use this analysis for identifying oscillating subcircuits which may be leveraged for *phase abstraction*.

The work of [2] proposed another use of this analysis: to identify *transient* signals which settle to a reducible (e.g., constant) behavior after several timesteps. The verification process may then be decomposed, leveraging bounded techniques to analyze the first several timesteps before the transients settle, then time-shifting the design and simplifying the transients for unbounded verification thereafter. This work is similar in spirit to the primary application domain considered in this paper: to enable the reduction of designs with intricate initial values. However, these are complementary techniques and we have found them both useful in conjunction.

Symbolic Trajectory Evaluation (STE) [3] is a related simulation technique. In STE, ternary simulation is combined with symbolic simulation, by encoding ternary value functions using a pair of BDDs or other *dual-rail* based expressions. Users specify assertions of the form $A \Rightarrow C$, where A is the *antecedent* that specifies the values with which to drive the simulation, and C is the *consequent* that specifies the expected results of the simulation. The advantage of STE over scalar simulation is that it can cover large input spaces efficiently and precisely. The complexity of STE is dependent on the number of symbolic variables in the the antecedent, not necessarily the size of the design, so it can scale to large designs such as complex datapaths and memory arrays. While using related analysis methods, our approaches are distinct in numerous ways. First, our application domain is approximate reachability analysis to facilitate model checking, wherein we do not have an antecedent to dictate where symbols should be introduced nor a consequent against which we may attempt to refine lossy X values. Our analysis is intended to efficiently facilitate subsequent verification algorithms, and without the heuristics described in this paper there may be blowups in runtime or memory if too many symbols are introduced, vs. too coarse of reachability approximation if inadequate symbols are introduced.

Our form of symbolic simulation uses the same value domain as *quasi-symbolic simulation* [4]. In quasi-symbolic simulation, value functions are restricted to the set $\{0, 1, X, X_A, \neg X_A, X_B, \neg X_B, \ldots\}$, where $\{X_A, X_B, \ldots\}$ are symbolic variables corresponding to netlist inputs. Instead of supporting arbitrary symbolic functions, quasi-symbolic simulation employs case-splitting to eliminate symbolic variables and remove conservatism (i.e., propagation of X to a checked output). In contrast, our technique does not seek a complete symbolic simulation of the netlist, and does not employ any case-splitting. Rather, we introduce symbolic variables selectively to enhance approximate reachability analysis. Also, our technique may introduce symbolic variables at gates internal to the netlist in addition to the netlist inputs, which often tightens the approximation.

Alg. 1. Approximate reachability with ternary simulation

```
 1: function approxReachability(design)
 2:     for all (primary inputs I in design) do I = X
 3:     for all (registers R in design) do R = X
 4:     ternarySimulate(design)
 5:     state = vector of register initial state valuations
 6:     seen = { state }
 7:     for time = 0; ; ++time do
 8:         Assign registers their corresponding values in state
 9:         ternarySimulate(design)
10:         state = vector of register next state valuations
11:         if (state ⊆ seen) then seen over-approximates the reachable states
12:         seen = seen ⋃ { state }
13:     end for
14: end function
```

III. Preliminaries

We consider gate-level sequential logic designs, and for convenience we assume the netlist is expressed as an And-Inverter Graph (AIG). That is, every gate in the design is either a constant, an AND gate, inverter, or primary input. We also consider *registers* which hold the state of the design. Registers have an associated next state function that defines their value in the next time step.

In our model, registers also have an explicit initial value function. For registers with a constant initial value, this function maps to the corresponding constant gate. For more complex initial values, this function maps to a a combinational subcircuit which is used to encode a set of initial states. Such complex initial values arise in many contexts. For example, they may be necessary in SEC applications to represent an arbitrary power-on state. Even for designs with simpler initial values, a more complex initial state may arise through a verification-enhancing transformation such as retiming [5] or temporal decomposition [2]. While the commonly used netlist format *AIGER* assumes that every initial value is constant-0 [6], more complex initial values are supported through synthesizing a multiplexor at the output of every latch which may drive the desired initial value at time 0.

Ternary simulation is a way to approximate netlist behavior. Inputs are assigned values in $\{0, 1, X\}$, and simple rules govern how these values propagate through the logic. We use ternary simulation on AIGs, and in this context `and(A,B)` is 0 if either `A` or `B` are 0. `and(A,B)` is X if either `A` or `B` are X. Otherwise, when `A` and `B` are both 1, `and(A,B)` is 1. We define inversion in the normal way, though `not(`X`)` = X.

Ternary simulation can be employed to perform approximate reachability analysis, as shown in Algorithm 1. Primary inputs and registers are initially assigned X values, and ternary simulation is used to derive the initial state values. A set of *seen* states is maintained, initially equal to just this ternary initial state. The algorithm then iterates, **(1)** assigning the current state values to registers while retaining the X values on primary inputs, **(2)** using ternary simulation to derive the next state values, and **(3)** using the set of *seen* states to detect convergence.

Convergence is detected through checking if the next state cube is contained in the set of *seen* states. There are several

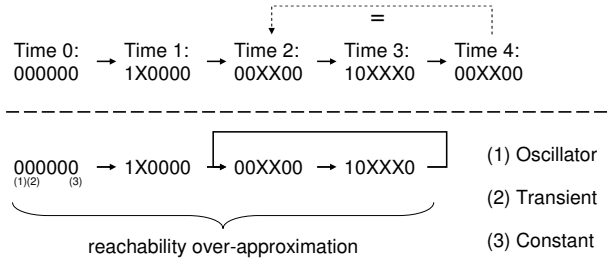

Fig. 1. Ternary simulation example for a design with six registers

ways this could be implemented. For example, BDDs [7] can be used to represent the set of seen state cubes, and the containment check can be implemented using BDD operations. In practice, the performance of such approach is prohibitive. Instead, we simply let the *seen* states be a list of ternary states, and instead of *state* $\subseteq$ *seen* we test if there exists an *s* in the *seen* list such that *state* $=$ *s*. Such a check can be implemented efficiently using a hash table, and this approximation to containment usually does not affect the convergence of Algorithm 1.

An example of reachability analysis with ternary simulation is shown in Figure 1. Five iterations are performed before it is determined that the time 4 state is equal to the time 2 state. This results in the approximate reachable state graph shown in the bottom half of the figure.

Ternary simulation-based reachability analysis has many applications. We implement this in a library which is used to characterize the netlist in various ways:

Oscillators: In Figure 1, register (1) is an oscillator, meaning that it periodically oscillates between 0 and 1 valuations. Designs that have oscillators may be simplified using phase abstraction [1], enabling significant verification benefits such as yielding a smaller netlist, enabling greater reduction potential through other transformations, and reducing diameter.

Transients: In Figure 1, register (2) is transient, meaning that after the initial time steps its value settles and remains constant forever after. Verification of designs that have transients can be simplified [2] through time-shifting their behavior, enabling reduction of the transient signals.

Redundancies: Gates that act as constants, or pairs of gates that are equivalent/antivalent in every reachable state, may be directly merged. In Figure 1, register (3) is constant, and in our implementation we replace this register with a constant-zero gate. Note that such a reduction may generally enable other reductions, such as constant propagation and cone-of-influence reduction.

IV. Symbolic Simulation

In this paper we strengthen reachability approximation by considering symbolic simulation as well as ternary simulation, similar to quasi-symbolic simulation [4]. In this section we define our notion of symbols and how they can be handled in simulation.

Symbols, written in the form X_A for some subscript A, represent concrete values that are not being precisely modeled. In contrast, the ternary X represents a *completely* unknown symbol. If two signals evaluate to X we conservatively conclude that the signals may not be equal, but if the signals both evaluate to X_A they are treated as equivalent.

We can expand ternary simulation to include a set of symbols $\{X_A, X_B, \ldots\}$ by letting signals take values in $\{0, 1, X, X_A, \neg X_A, X_B, \neg X_B, \ldots\}$. We retain the traditional ternary simulation evaluation from Section III for conjunction, with rules listed in order of precedence:

0 Identity: If $a \equiv 0$ then $a \cdot b = 0$. Likewise for $b \equiv 0$.
1 Identity: If $a \equiv 1$ then $a \cdot b = b$. Likewise for $b \equiv 1$.
X **Identity:** If $a \equiv X$ then $a \cdot b = X$. Likewise for $b \equiv X$.

If none of the above rules apply then both signals are symbolic: $a = X_A$, $b = X_B$. We apply the following symbolic rules:

Idempotence: If $X_A \equiv X_B$ then $X_A \cdot X_B = X_A$.
If $X_A \equiv \neg X_B$ then $X_A \cdot X_B = 0$.
Peephole Optimization: If $X_B \equiv X_A \cdot X_Z$ then $X_A \cdot X_B = X_B$.
Hashing: If $X_A \cdot X_B$ is in the hash table, return the previously-stored result.
New Symbol: Create a symbol X_C to represent $X_A \cdot X_B = X_C$. Store X_C in the hash table.

First idempotence is used to handle cases where a symbolic value is conjoined with itself. Next, peephole optimization is used to find cases of nested conjunctions with shared arguments. A hash table is used to determine if the result of $X_A \cdot X_B$ was previously computed. If all other checks fail then we create a new symbol X_C to represent the result of a conjunction.

V. Incorporating Symbols Into Reachability

In this section we discuss how to incorporate symbols into approximate reachability analysis. Symbolic simulation offers greater resolution than ternary simulation, but symbols must be applied judiciously. If every signal was handled symbolically then Algorithm 1 would perform exact reachability analysis – though likely with an explosion in the number of symbols represented, leading to unacceptable runtime or memory consumption.

We are motivated by designs that have complex initial values, and we would like to use symbolic simulation to represent these initial values. In our model, the initial values are derived from combinational functions over the primary inputs. For this reason, we assign the inputs symbolic values and trust that these symbols will propagate to the initial values, allowing us to represent these initial values more precisely.

One risk is that the number of symbols can explode. Specifically, new symbols are introduced for each primary input and with each application of the *New Symbol* rule of Section IV. If the current state of the design contains one new symbol for each step of approximate reachability, then Algorithm 1 can never converge. We limit the number of

Alg. 2. Approximate reachability with ternary and symbolic simulation

```
 1: function approxReachability_symbols(design)
 2:     for all (primary inputs I in design) do I = new symbol
 3:     for all (registers R in design) do R = X
 4:     symbolicTernarySimulate(design)
 5:     state = vector of register initial state valuations
 6:     seen = { state }
 7:     for time = 0; ; ++time do
 8:         Assign registers their corresponding values in state
 9:         if (time = 0) then
10:             symbolicTernarySimulate(design)
11:         else
12:             for all (primary inputs I in design) do I = X
13:             symbolicTernarySimulate_noNewSymbols(design)
14:         end if
15:         state = vector of register next state valuations
16:         if (state ⊆ seen) then seen over-approximates the reachable states
17:         seen = seen ⋃ { state }
18:     end for
19: end function
```

symbols by only handling primary inputs symbolically at time 0. In addition, we only allow the *New Symbol* rule to apply in time 0. At all other times, we consider $X_A \cdot X_B = X$.

Algorithm 2 illustrates our framework for approximate reachability using ternary and symbolic reachability. We create new symbols to represent primary inputs at time-0, and at all other times we assign primary inputs the value X. Algorithm 2 utilizes two simulation routines, *symbolicTernarySimulate* and *symbolicTernarySimulate_noNewSymbols*. The function *symbolicTernarySimulate* is used only at time-0 and simulates the design as described in Section IV. We have found it effective in our desired application domain to *not* introduce any additional symbols after time 0, and this is accomplished by using the function *symbolicTernarySimulate_noNewSymbols* which implements the methods of Section IV but treats $X_A \cdot X_B = X$ when the value of this conjunction cannot be otherwise determined through idempotence, peephole optimization, or hash lookup.

Algorithm 2 introduces new symbols only at time-0. For all time > 0 no additional symbols are created, but the symbols created at time-0 can continue to propagate through the logic at later times.

By restricting the application of symbolic simulation, we derive an approximate reachability algorithm that retains most of the performance of Algorithm 1 but is significantly more precise for numerous classes of important verification problems. This increased precision allows the efficient characterization of reachability information – constant or equivalence signals, oscillators, and transients – that otherwise are undetectable.

VI. Generalized Simplification using Symbols

Algorithm 2 returns an over-approximation to the set of reachable states. Our implementation uses this information in several application domains, some of which become more complex when the reachability information contains symbols. In particular, the simplification of symbolic constant gates, oscillating registers, and transients is affected.

Symbolic constant gates are those that always evaluate to the same symbolic value in every reachable state. Such gates may be simplified by replacing them with a subcircuit that represents the given symbol.

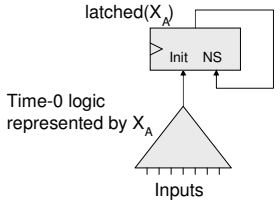

Fig. 2. A subcircuit that represents the symbol X_A

When oscillating registers are identified, we use phase abstraction to simplify the netlist. Phase abstraction will unfold the transition relation modulo a detected periodicity, and simplify the netlist by injecting constant values in place of the corresponding oscillator. However, when the reachability information contains symbols, the detected oscillators may assume symbolic periodic behavior. For example, we have observed period-two oscillators with signature $0, X_A, 0, X_A, \ldots$. To simplify such symbolic oscillators, we must replace a register in the unfolded transition relation with a reference to a subcircuit that represents the given symbol.

When transients are identified, temporal decomposition is able to simplify the design by time shifting and replacing each transient gate with the corresponding redundant value to which it settled. When using symbols, in cases we find transients that settle to a symbolic values. Temporal decomposition can simplify these by replacing each transient with a reference to a subcircuit that represents the given symbol.

Recall that in Section V we use symbols to represent the value of inputs, or combinational functions thereof, at time 0. We may obtain a subcircuit that represents such a symbol by simply latching the time-0 value; we fabricate a register whose initial value is that corresponding signal, and whose next-state function holds its current value. This corresponding logic may be used in the above three application domains to simplify the netlist, extending our ability to simplify a netlist using the enhanced reachable-state characterization enabled through using symbols. This logic depicted in Figure 2 shows a subcircuit that represents the symbol X_A.

Simplifying logic using this procedure is often beneficial to reduce overall netlist size and verification complexity. However, this procedure does entail synthesizing registers, which may be undesirable in cases. Our implementation uses several heuristics to minimize this impact: **(1)** when multiple two subcircuits $\mathrm{latched}(X_A)$ and $\mathrm{latched}(X_B)$ are synthesized, we try to share registers across these two symbol representations, and **(2)** we disallow simplifications in cases where the total number of registers would increase.

VII. Accelerating Convergence with Saturation

Approximate reachability, shown in Algorithm 1, is an iterative procedure which successively explores sets of states until it detects a fixedpoint. For deep and complex industrial designs, the number of iterations required for convergence may be prohibitive. Although we improve on the precision

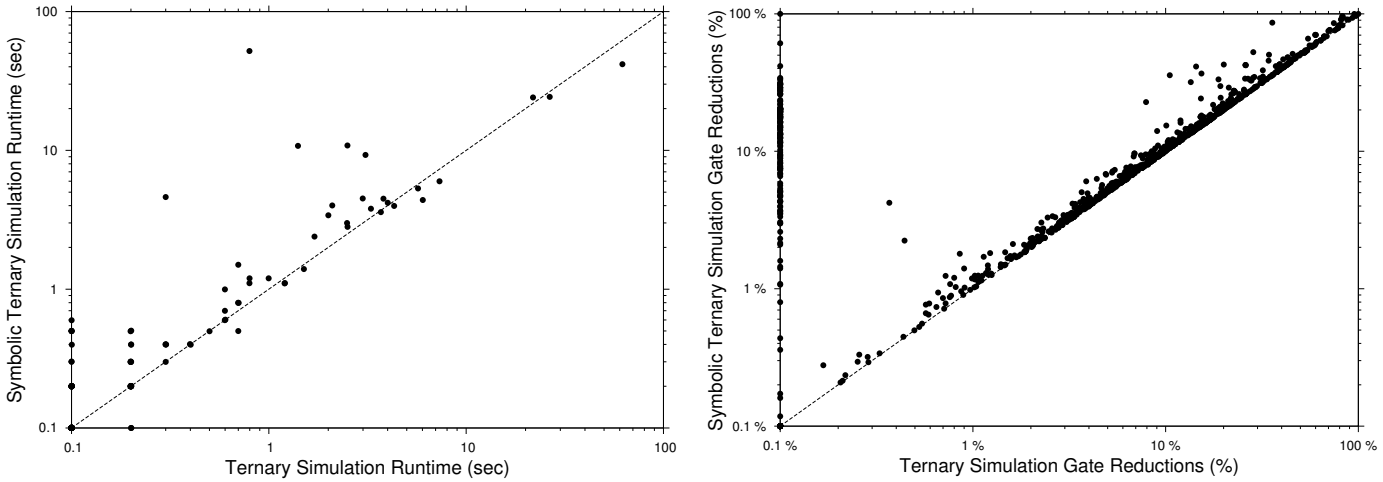

Fig. 3. Approximate reachability runtime on the IBM designs. Left: runtime, Right: gate reductions

Alg. 3. Approximate reachability with X-saturation

```
1: function approxReachability_symbols_saturation(design, cycleLimit)
2:     setup approximate reachability as in Algorithm 2, lines 2-6
3:     for time = 0; ; ++time do
4:         Assign registers their corresponding values in state
5:         if (time ≥ cycleLimit) then
6:             for all r ∈ {registers not oscillating or constant} do r = X
7:         end if
8:         do one iteration as in Algorithm 2, lines 9-17
9:     end for
10: end function
```

of approximate reachability by using symbols in Algorithm 2, this convergence problem remains and in cases worsens due to the extra precision. In this section we detail our solution to accelerate convergence when a pre-determined resource threshold is exceeded.

The approximate reachability loop does not converge if new register valuations are encountered, and we can accelerate convergence by limiting the register valuations. Specifically, it is always conservative to further over-approximate the computation by overwriting register valuations with X. We refer to this process as X*-saturation*. If all registers are assigned X then the reachability approximation process will immediately converge; however, it would contain no useful information. The difficulty with effective X-saturation is thus in determining which subset of registers to X-saturate, and when to perform this saturation, balancing precision vs. runtime.

As in Section VI, approximate reachability may be used to detect oscillating registers and constants which may be used for phase abstraction and redundancy removal. Furthermore, we have found that such gates are high-fanout registers that influence much of the netlist behavior. We are motivated to always preserve constants and oscillating registers by not X-saturating them because **(1)** such saturation would limit results that are useful for phase abstraction and redundancy removal, and **(2)** X values injected on such gates would quickly propagate through the netlist and dramatically weaken the resulting reachability approximation.

Algorithm 3 is an extension of Algorithm 2 which includes X-saturation. It takes one additional argument *cycleLimit* which is the number of iterations that are allowed before registers are X-saturated. After *cycleLimit* is exceeded, non-oscillating and non-constant registers are forced to have the value X. This further approximation causes the algorithm to converge quickly, with the total number of cycles usually being only slightly larger than *cycleLimit* in practice.

Algorithm 3 requires constant and oscillating registers to be detected. In our implementation, we efficiently identify oscillators using a sliding window technique which is able to identify oscillators with period ≤ 128. In addition, we also detect *delayed oscillators* which may assume variable behavior during the design's initialization phase but thereafter act as oscillators. Both true- and delayed-oscillators influence large sub-circuits in the netlist, and to preserve a useful reachability approximation it is vital to not X-saturate these registers.

VIII. Experimental Results

All techniques described in this paper have been implemented in the IBM internal verification tool *SixthSense* [8]. In these experiments we utilize two benchmark sets:

IBM SEC: We use a suite of 1122 challenging industrial SEC problems. These designs come primarily from high performance microprocessors, and the largest such design has 5.3M AIG AND gates and 330k registers. In our framework, pairs of registers are often initialized with the same non-deterministic random value in order to check equivalence modulo any initialization sequence.

HWMCC'10 SEC: To enable evaluation against publicly available benchmarks, we evaluated our techniques against a subset of benchmarks from the Hardware Model Checking Competition (HWMCC) [9]. However, none of these benchmarks directly exhibits the complexities often faced in industrial SEC benchmarks. We thus emulated the industrial challenges in these problems as follows:
(1) We simulated each design for 1000 cycles starting from the initial state, and inferred register equivalences from the simulation data.

Benchmark	Reg. Equivalences	Gates	Ternary Simulation			Symbolic Ternary Simulation		
			Time	Iterations	Gates Reduced	Time	Iterations	Gates Reduced
139464p22_sec	1	20768	0.0	6	2	0.0	6	2
139464p23_sec	1	20800	0.0	6	2	0.0	6	2
139464p24_sec	1	20832	0.0	6	2	0.0	6	2
bob1u05cu_sec	17	51650	0.0	37	29627	0.0	37	29813
bobmitersynbm_sec	1507	47416	0.0	7	0	0.0	8	4385
bobsmmem_sec	1068	42351	0.0	7	0	0.0	12	2884
mentorbm1and_sec	28	46286	0.1	37	4279	0.1	37	7190
mentorbm1p02_sec	19	51450	0.0	37	8521	0.0	37	10662
mentorbm1p03_sec	19	51447	0.1	37	8525	0.0	37	10664
mentorbm1p04_sec	21	51444	0.1	37	8442	0.1	37	10599
mentorbm1p05_sec	21	51351	0.1	37	8060	0.1	37	10279
mentorbm1p07_sec	28	46478	0.1	37	4921	0.0	37	7724
mentorbm1p08_sec	19	51435	0.1	37	8511	0.1	37	10658
mentorbm1p09_sec	19	51450	0.1	37	8525	0.1	37	10666
mentorbm1p10_sec	19	51447	0.0	37	8525	0.0	37	10664
mentorbm1p12_sec	19	51429	0.1	37	8462	0.1	37	10601
pj2006_sec	1	37411	0.0	6	1826	0.0	6	1826
pj2013_sec	1	37518	0.0	7	2134	0.1	7	2134
pj2015_sec	1	41424	0.0	7	2076	0.0	7	2076
pj2017_sec	1	41580	0.0	7	1315	0.0	7	1315
Average Performance					1.00			1.18

Fig. 4. Detailed comparison of approximate reachability-based design simplifications on the HWMCC'10 SEC benchmarks

(2) For each register equivalence class, we constructed a new primary input to model the non-deterministic initial value for that class. For all registers in this class, we replaced the initial value with the new primary input.

We have made the modified HWMCC benchmarks publicly available [10].

In our experiments we simplify each design by applying approximate reachability in order to find constant and equivalent signals. This emulates the default flow in *SixthSense* where approximate reachability is the first algorithm applied to a design under verification, due to its speed, scalability, and capability to significantly simplify the design for subsequent more-precise analysis. We repeat this flow twice: once using only ternary simulation, and again using techniques presented in this paper. All experiments were run on a cluster of 4 GB, 2 GHz POWER5 workstations.

A. IBM SEC Results

Figure 3 examines the performance of reachability analysis with symbolic and ternary simulation on the IBM SEC benchmarks.

The first plot in Figure 3 shows the runtime of approximate reachability. Introducing symbolic simulation almost always slows approximate reachability, though this slowdown is negligible with most runs completing in less than 10 seconds. Given the large sizes of these industrial benchmarks (up to 5.3M ANDs), we are satisfied with this minimal runtime overhead.

The second plot in Figure 3 shows reductions enabled using the corresponding approximate reachability information. We show the number of gates eliminated during design simplification as a percentage to the original number of gates. Most designs see greater reductions with symbolic simulation. In addition, many designs were not simplified at all with ternary simulation but now are simplified with ternary and symbolic simulation. In some cases, symbolic simulation adds sufficient resolution to prove the properties outright.

B. HWMCC'10 SEC Results

Figure 4 examines the performance of reachability analysis with ternary and symbolic simulation on the HWMCC'10 SEC benchmarks. Recall that these benchmarks were created from the HWMCC'10 benchmarks by finding suspected equivalent registers and transforming their corresponding initial states. Column 2 shows the number of equivalent register pairs that were found during that process.

In Figure 4 we can see that introducing symbolic simulation did not affect the runtime of approximate reachability in any measurable way.

Next examine the number of iterations. This is the number of time steps processed by approximate reachability before convergence. We expect this number to increase when symbols are utilized due to the more precise state representation. However, using our methods of minimally-introducing symbols, the additional number of iterations imposed is minimal, and usually enabling symbolic simulation does not increase the number of iterations at all.

Figure 4 also shows the number of gates that were reduced through approximate reachability-based design simplification. Enabling symbolic simulation allows for more gate reductions, 18% on average. In cases, approximate reachability is unable to simplify the design without the additional resolution provided by symbolic simulation.

C. Saturation Results

Figure 5 examines X-saturation. On the combined set of 1200 benchmarks, 21 benchmarks (1.75%) failed to converge within 512 iterations. All of these benchmarks are IBM designs (the HWMCC designs converged quickly), and are labeled ibm1 through ibm21 in the table.

As a baseline, we consider Algorithm 2 which is limited to 1200 seconds and 1M iterations. Eight of our designs hit one of these limits, and approximate reachability stopped abruptly with no useful reachability information with which to reduce

		Baseline			With Saturation Heuristics		
Benchmark	Gates	Time	Iterations	Gates Reduced	Time	Iterations	Gates Reduced
ibm1	15853	1.1	4120	1052	0.2	519	900
ibm2	2507	5.9	49153	937	0.1	569	348
ibm3	25805	725.0	1000000	-	0.6	631	1563
ibm4	30058	763.1	1000000	-	0.4	521	2102
ibm5	11686	291.6	1000000	-	0.1	521	281
ibm6	13760	68.5	131101	2787	0.2	516	2438
ibm7	35377	180.8	131299	3148	0.7	519	3056
ibm8	30056	0.5	551	2616	0.5	523	2416
ibm9	916576	42.1	1060	66443	21.8	540	63849
ibm10	11802	55.0	196612	10	0.2	522	10
ibm11	2697713	1200	6320	-	105.6	561	1620533
ibm12	626060	154.6	32920	63693	2.5	532	60402
ibm13	11686	295.9	1000000	-	0.2	521	281
ibm14	1698160	35.0	525	1367988	33.7	521	1366566
ibm15	760354	18.8	525	601602	19.0	521	601598
ibm16	30056	0.5	551	2616	0.5	523	2416
ibm17	385244	3.0	523	219	3.4	523	186
ibm18	395964	10.9	523	7375	10.1	523	7372
ibm19	242756	1200	163470	-	4.0	516	0
ibm20	411849	1200	93004	-	6.7	518	17731
ibm21	20229	1200	490270	-	1.2	544	4456
Average Performance		1.00		1.00	0.33		0.94
Cummulative Performance		1.00		1.00	0.03		1.77

Fig. 5. X-Saturation on the IBM benchmarks

the size of the design. The remaining 13 designs converged and the resulting reachability approximation was effective at reducing the design size, though the runtimes were very long.

The final columns in Figure 5 show Algorithm 3 with *cycleLimit* set to 512. That is, a subset of the registers are X-saturated starting at iteration 512. In most cases, this allows the algorithm to converge with just a few extra iterations. On the 13 designs that previously converged with a high number of iterations the runtime is reduced by 67%. X-saturation does compromise the resolution of the reachability approximation slightly, though on this subset of designs we preserve 94% of the reductions, on average.

It is most interesting to examine X-saturation on the 8 designs that previously hit either the 1200 second limit or the 1M iteration limit. With X-saturation, approximate reachability converges on all of these designs, and the algorithm is very fast. In all cases, the reachability approximation is suitable for design reductions, and our best example is ibm11 where approximate reachability is able to reduce the design size by 60%. Without X-saturation we cannot realize such reductions.

Cummulatively we are able to reduce the runtime by 97% while increasing the reductions by 77%, considering the designs that previously had no reductions because of computational resource limits.

IX. Conclusion

In this paper we enhance techniques for approximate reachability analysis using ternary simulation in two ways:

- We use ternary simulation enhanced with symbols to allow greater precision in modeling registers that have complex initial values. The more precise reachability approximation enables considerably greater reduction opportunity, especially on industrial SEC models.
- We introduce X-saturation as a way to force approximate reachability into convergence on complex industrial designs. In cases of slow convergence, this helps to dramatically reduce the runtime. In cases of no convergence, this helps to provide a useful reachable state approximation where previously we had none.

We additionally introduce extensions to ternary simulation-based application domains of redundancy removal, phase abstraction, and transient elimination to generalize them accordingly given our symbolic techniques. All of these techniques are implemented in IBM internal verification tool *SixthSense*, and we have found them to be indispensable on large and complex industrial SEC problems.

References

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

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

[3] C. johan H. Seger and R. E. Bryant, "Formal verification by symbolic evaluation of partially-ordered trajectories," in *Formal Methods in System Design*, 1993.

[4] C. Wilson and D. L. Dill, "Reliable verification using symbolic simulation with scalar values," in *DAC*, 2000.

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

[6] A. Biere, "The AIGER And-Inverter Graph (AIG) format, version 20070427." http://fmv.jku.at/aiger/FORMAT-20070427.pdf.

[7] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, August 1986.

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

[9] A. Biere and K. L. Claessen, "Hardware Model Checking Competition (HWMCC) 2010 benchmarks." http://fmv.jku.at/hwmcc10.

[10] M. L. Case, "Sequential equivalence checking variants of the Hardware Model Checking Competition 2010 benchmarks." http://case-home.com/publications/hwmcc10_sec.tgz.