Effective Liveness Verification using a Transformation-Based Framework

Pradeep Kumar Nalla*, Raj Kumar Gajavelly*
*IBM Systems and Technology Group, Bangalore, India
{pranalla, rgajavel}@in.ibm.com

Hari Mony†, Jason Baumgartner†, Robert Kanzelman‡
†IBM Systems and Technology Group, Austin, TX
‡IBM Systems and Technology Group, Rochester, MN
{harimony, baumgarj, bobkanz}@us.ibm.com

***Abstract*—Liveness properties such as *"will every request eventually get a grant?"* are crucial to the verification of a variety of design types. Liveness properties may only be falsified by infinite-length counterexamples, represented using lasso-shaped traces with a prefix (e.g., showing a particular request) followed by a repeating suffix loop (e.g., showing no grant). A variety of techniques have been developed to solve liveness properties, including BDD-based model checkers, various bounded liveness checking methods, and liveness-to-safety conversion. Nonetheless, the verification of such properties is computationally very challenging; no single algorithm works best, and many industrial-sized liveness problems remain practically unsolvable.**

In this paper, we detail three approaches to solve liveness properties in our verification toolset *SixthSense*. The first is in the context of dynamic verification, enhancing a simulation engine to support native liveness checking. The second approach is formal, using abstraction-guided liveness-to-safety conversion for greater scalability. The third approach leverages complementary transformation algorithms to enhance the scalability of liveness checking algorithms. We additionally address automated verification of liveness properties on designs with memories, which has not received significant prior research. Experiments are provided to confirm the effectiveness of our techniques.

I. Introduction

Verification is the process of establishing the functional correctness of a design. The principal industrial verification methods are *simulation* and *formal verification*. Simulation checks whether the design exhibits proper behavior under a series of functional tests, whereas formal verification uses proof techniques to establish the correctness of a design with respect to a formal specification under any possible execution.

Verification properties may be classified as *safety* vs *liveness*. Safety properties assert that something bad never happens. In Linear Time Temporal Logic (LTL), safety properties are commonly expressed in form: $G\neg\phi$. Liveness properties assert that something good will eventually happen. In LTL, liveness properties are commonly expressed in the form: $GF\phi$ or $G(\varphi \rightarrow F\psi)$.

Liveness verification is becoming increasingly prevalent as designs grow in scale as a consequence of Moore's Law: more cores, more complex shared communication networks, and more intricate memory subsystems all mandate more sophisticated transaction queuing and arbitration, which intrinsically entail liveness properties. While liveness checking and safety checking are in the same complexity class, liveness checking is known to be significantly less scalable in practice despite the variety of algorithms available to solve them.

Liveness verification often involves the use of *fairness constraints*, which restrict the set of valid counterexamples by enforcing certain conditions to hold in the lasso loop. For example, given a system with various priorities associated with requests, one may need to enforce that higher-priority requests eventually subside to allow the system to grant lower-priority requests, without which liveness checking of lower-priority requests would spuriously fail.

The rest of the paper is organized as follows. Section II provides background formalisms. Section III describes some background techniques for liveness verification. In Section IV we outline our technical contributions: **1)** an efficient simulation-based liveness checking algorithm which supports memories; **2)** an abstraction-refinement technique to create smaller liveness-to-safety converted problems; and **3)** an evaluation of a variety of transformation and proof techniques for solving complex liveness problems. These contributions are evaluated experimentally in Section V.

II. Preliminaries

We focus on the verification of hardware designs, which may be modeled as Finite State Machines (FSMs).

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.

The design constructs which define FSM states are bit-level state elements hereafter referred to as registers, as well as memories such as RAM.

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. A trace illustrating a refutation of a property is referred to as a *counterexample*.

While safety property counterexamples are finite-length, liveness counterexamples must be infinite-length. Practically, it is necessary to represent traces using finite data structures. This may be achieved by referencing a special *LOOP* input which non-deterministically initializes, and at some point asserts and remains asserted for the duration of the finite trace. Semantically, such a trace is interpreted to mean that the suffix of the LOOP assertion may be indefinitely repeated as a lasso loop to constitute a valid infinite-length trace: the next-state of the design at the end of the loop matches the current state at the beginning of the loop.

Definition 3: *Fairness constraints* are used to restrict the set of possible counterexamples to liveness properties. In particular, a liveness property counterexample must illustrate every fairness constraint satisfied at least once in its lasso loop.

III. Existing Liveness Checking Techniques

A. BDD-Based Model Checking

Symbolic model checkers based upon BDDs (Binary Decision Diagrams) constituted the first breakthrough of relative scalability for formal hardware verification. Most early model checkers supported expressive temporal logics such as LTL or CTL. A good reference describing various BDD-based liveness checking algorithms is provided in [12]. BDDs are capable of representing large state sets compactly, scaling to designs with up to several hundred state elements in cases. However, BDD size may grow exponentially with respect to design size, hence BDD-based model checking does not scale up to large designs.

B. Liveness-to-Safety Conversion (LTS)

Recall that liveness properties have infinite-length counterexamples, which entail a repeating lasso-loop suffix in which the next-state at the end of the loop matches the current state at the beginning of the loop. It is noted in [14] that one may convert a liveness problem to a safety problem by adding suitable counterexample-detection logic. In particular, the conversion uses a shadow state element r_s for every original state element r to non-deterministically select when to sample the design state and thereby check for a state repetition loop, during which the behavior of the liveness and fairness conditions are observed as constituting a valid counterexample scenario. A major advantage of converting liveness-to-safety is that it enables the applicability of a large number of safety checking algorithms. However, this translation substantially increases problem size, and in practice tends to yield problems which are not amenable to several popular SAT-based proof techniques.

C. Bounded Liveness Checking

Instead of directly attempting to prove the absence of an infinite-length liveness failure, it has been noted that one may instead attempt a finite bounded proof [14]. For example, given a property $GF\phi$, instead of proving that $\neg\phi$ cannot occur infinitely often, one may check that $\neg\phi$ cannot occur after more than k time steps. This check can be much more scalable than LTS since it does not entail the overhead of the state-repetition logic, and for finite designs it is complete given an adequately-large bound. Though the bound may be exponential with respect to design size in the worst case, hence this approach is not practical for certain designs.

This technique was improved in [7], which instead of checking that $\neg\phi$ cannot occur after more than k time steps, the technique counts the maximum number of times the specified failure condition $\neg\phi$ may occur. It performs a sequence of safety queries as necessary to find a large-enough bound to avoid spurious failures. An additional contribution of that work is the generation of additional fairness constraints, which are signals in the design that, once asserted, remain asserted forever. Unlike traditional fairness constraints, these constraints cannot affect whether the given liveness properties pass or fail. Though similar in spirit to the generation of semantics-preserving reachability constraints in [6] for the purpose of enhancing certain downstream verification algorithms, these fairness constraints practically enable proof techniques such as IC3 [5] to converge with a small bound. However, for large industrial size designs, it is observed that the extraction of stabilization constraints itself may consume significant resources. In addition, it is noted that this method is best suited for passing vs. failing properties.

IV. Contributions

The primary goal of this paper is to enhance the scalability of liveness verification for large designs. In this section, we describe our three contributions to this goal: **1)** Enhancing liveness verification in the context of simulation including designs with native memories. **2)** Abstraction-guided liveness-to-safety conversion. **3)** Effective application of transformation-based verification for liveness properties.

A. Simulation-Based Liveness Checking

Formal verification provides exhaustive coverage and has the power to uncover subtle bugs, in contrast with traditional simulation. However, for large industrial designs, formal verification in practice often suffers incomplete coverage due to capacity limitations. This capacity challenge is worsened if the design has large memories. To our knowledge, no prior work has addressed the topic of efficient liveness checking in a simulation environment for designs with memories. While the focus of this work is primarily formal, we note that for very large designs or designs with very deep fails, simulation may offer the best chance of exposing a flaw. Furthermore, coupled with semi-formal extensions such as use of bounded model checking (BMC) from simulation-explored states, rarity-based simulation, and abstraction-guided simulation, an efficient simulation framework is practically an essential component of a robust verification toolset. Figure 1 delineates our simulation-based liveness checking algorithm.

As a preprocessing, we enumerate the registers in the fanin of a livelock property as well as the fairness constraints, which are assumed to be synthesized as gates [1]. The function `collectImportantStateElts` in Figure 1 performs this operation. We use this set to narrow the scope of the state repetition check, noting that this cone-of-influence reduction is known to preserve the semantics of liveness checking and practically yields easier-to-debug counterexamples [3].

In each simulation step, we check whether the current state has been previously visited using a hash table. The use of the hash table avoids quadratic slowdown where each simulated time step j must be compared to each prior time step $i < j$. If the state has already been visited, we next check if the liveness condition has exhibited a failing behavior during that loop – e.g., given liveness property $GF\phi$, we check whether $\neg\phi$ has held during the loop. Additionally, we check whether all fairness constraints have been satisfied during that loop. This is done using function `checkAsserts`. If these conditions hold, we report a valid liveness hit with the discovered lasso-shaped loop suffix.

As an optimization, when a state repetition is observed which does not constitute a valid counterexample, we only

```
verify_Liveness(l, F) {
  // l = liveness gate
  // F = set of fairness gates
  collectImportantStateElts(l, F);

  for each simulation step i {
    regState = formRegStateUsingImpRegisters();
    visit_before =
      addStateInStHash(regState, i, &loop_start);

    if(visit_before) { // loop condition
      if (checkAsserts(loop_start, i)) {
        report failure;
        break;
      }
    }
  }
}
```

Fig. 1. Simulation-based liveness checking algorithm

store the earliest occurrence of that state in the hash table as this is adequate to detect all counterexamples. Another optimization is to speed up the liveness-failure check and fairness satisfaction check by maintaining a record of the most-recent time steps at which each of these has been observed. Since we only consider loops ending with the current simulation step, this data is adequate to capture all counterexamples and avoids potential quadratic slowdown due to redundantly checking earlier time steps.

Discussion on arrays: If the design has arrays, they must be considered when checking for a state repetition. We note that, for designs with large memories, it is prohibitively expensive to model *every* row. Indeed, the goal of modeling memories as arrays vs. bit-blasted registers is to attain greater scalability. An alternative is to represent state of the array including a counter indicating the number of writes that have occurred, along with information on rows that have been written with non-initial values. If it is desired to support non-deterministically-initialized arrays, array reads to unwritten rows may also need to be recorded as those define data values. The modifications necessary to support arrays in this process are as follows:

1) We also enumerate arrays in fanin of liveness and fairness gates during `collectImportantStateElts`.
2) We perform state hashing only on registers, and check for a repetition of array contents secondarily if the register state matches. If the register state matches, we first check if the number of array writes are the same between the register loop start and end for each important array. If so, this implies an array content match. Otherwise, we need to compare array contents for each array which logged more writes at the register loop end vs. start to see if the data contents actually differ. Note that we only record writes to the array, hence this check is $O(|writes|)$ vs. $O(|rows|)$. We additionally record writes for each array separately to make this check more efficient.
3) Finally, given a register + array match, we check if all fairness and liveness-failure conditions have been satisfied in this loop and if so, we report the liveness violation.

B. Abstraction-Guided Liveness to Safety (LTSABS)

As discussed in Section III-B, the process of liveness-to-safety conversion entails adding a shadow register for each original register to enable a precise state repetition check. In addition, designs with the memories further complicate this conversion requiring an explicit shadowing of all array rows in addition to logic to compare every row of the original vs. shadow arrays, which for automated hardware model checkers is practically as much overhead as bit-blasting the arrays to registers prior to liveness-to-safety conversion.

This doubling of the number of state elements tends to significantly burden verification algorithms, in many cases resulting in an unsolved property. It has been noted in past work that subsetting the set of shadow registers is a sound yet incomplete abstraction: a proof that no bad loop exists over a subset of state elements clearly implies that no bad loop exists over all state elements [13]. However, a counterexample obtained using a subset may not be valid for the entire design. This is similar to the observation that an exact liveness-to-safety conversion followed by localization abstraction which is able to cut-point shadow registers is sound yet incomplete [3]. While past work has noted that such an abstraction is possible, we are unaware of any published attempts to implement a well-tuned native liveness-to-safety abstraction-refinement loop. We have implemented this technique as per the following pseudo-code, and demonstrate its significant benefits in Section V.

1) Start with an initial subset (possibly empty) of state elements to include in the abstraction.
2) Perform liveness-to-safety conversion only including the specified subset of shadow registers.
3) Perform verification engine to see if the liveness property is solved. Proofs are valid by construction; counterexamples must be analyzed for validity using the following step.
4) If a spurious counterexample is detected, refine the abstraction by determining a set of state elements not shadowed in Step 2 to include in the next subset. Repeat Step 2.

Similar to state-of-the-art localization algorithms, a variety of techniques may be used to produce as few abstractions as possible which are small in size and immune to already-encountered spurious counterexamples. For example, proof-based abstraction-refinement may be used to analyze bounded BMC proofs over the non-abstract problem to determine a boundedly adequate set of shadow registers, i.e. those relevant to refuting a concrete counterexample of a given bound. Counterexample-based abstraction-refinement may be used to determine a set of missing shadow registers which cannot be made equivalent at the start vs. end of the spurious counterexample loop. Note that this type of abstraction is irrelevant to the prefix of the counterexample.

C. Scalable Liveness Checking via Transformations

Despite the availability of numerous useful liveness proof and falsification algorithms, for the most complex industrial problems, these algorithms are not sufficient alone to yield conclusive results. In practice, it is often beneficial to use a synergistic sequence of abstraction and reduction techniques to reduce the size of a verification problem before it becomes tractable for a core proof or falsification algorithm. The utility of such a transformation-based approach for liveness verification was also noted in [3], though the focus of that paper was on algorithmic extensions to enable the use of existing transformations on designs with liveness properties. One

contribution of this paper is to provide broader experimental evidence of the value of these transformations, and also to contrast the effectiveness of a variety of core liveness checking algorithms.

Some of the algorithms we have found particularly powerful in liveness proofs include the following:
BRN: A redundancy removal engine which uses combinational optimization techniques such as structural hashing and resource-bounded BDD- and SAT-based analysis to identify gates which are functionally redundant across all states [9], as well as a variety of rewriting techniques such as [10] to reduce design size.
MOD: A phase abstraction engine, which unfolds next-state functions modulo some depth k to abstract certain clocking and latching schemes [4].
EQV: A sequential redundancy removal engine [11].
RET: A min-area retiming engine [8], which reduces the number of registers by shifting them across combinational gates.
CUT: A reparameterization engine [2], which replaces the fanin-side of a cut of the netlist graph with a trace-equivalent, yet simpler, piece of logic.
LOC: A localization engine [2], 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.
ARY: An array expansion engine, which bit-blasts arrays into simpler registers.

In addition to above transformation algorithms, we have experimented with native liveness proof techniques including BDD-based model checking (an Emerson-Lei implementation as discussed in [12]) and k-liveness [7]. We also have used safety proof techniques subsequently to liveness-to-safety conversion including induction, interpolation and IC3.

V. Experimental Results

In this section we present several sets of experiments. The first set compares the results using various liveness checking proof techniques with only light-weight prior simplification on the HWMCC 2012 benchmarks [1]. The second set describes results on industrial liveness testbenches. The last set of experiments demonstrates the efficiency of our simulation based liveness checking. All our algorithms were implemented in the transformation-based verification tool *SixthSense* [8]. Each experiment has a time limit of 3 hours and was run on a 4-core 8GB Linux machine.

A. Liveness Algorithm Comparison

Table I shows the results of various liveness algorithms on difficult HWMCC 2012 liveness benchmarks for which k-liveness took more than 1200 seconds, both with precise (**LTS**) and abstraction-guided (**LTSABS**) liveness-to-safety conversion. After conversion, we leverage several algorithms in parallel with prior application of light-weight Boolean simplification (BRN): IC3, BMC, interpolation (ITP) and localization followed by BDD-based reachability (LOC). We also ran k-liveness [7] and Emerson-Lei BDD-based model checking without conversion to safety [12]. In is noteworthy that we do not include induction after safety conversion in these experiments: practically, induction is ineffective on problems which cannot be easily boundedly solved, since even unique-state constraints are too weak over shadow registers to eliminate unreachable bad states.

The first and second columns indicate the design name and size in terms of the number of registers. The third column shows the known result. The fourth and fifth columns show the results of precise liveness-to-safety conversion, and columns 6-10 show the results of abstraction-guided conversion. Columns 11 and 12 show the results of bounded-liveness and BDD-based model checking algorithms, respectively. For the safety-converted experiments, the winning engine results are detailed in columns 5 and 7. For LTSABS, we indicate the number of refinements and the final number of shadowed registers in the abstraction in columns in 8 and 9. *T.out* indicates timeout. A simple Boolean optimization engine (BRN as discussed in Section IV-C) was used to preprocess all of these runs. Column 10 indicates percentage of reduction achieved in terms of number of registers for which we constructed shadow equality logic, i.e. with base line as $2\times$ the total number of registers available in the design.

Note that *arbixs08p03* is uniquely solved by BDD-based model checking; BMC is almost always the winner for failing properties; and IC3 is the predominant proof technique. Additionally, these results clearly indicate the value of LTSABS over LTS in terms of producing smaller designs for the final verification algorithm, aside from the *cujc** and *arbi0s32bugp03* examples which cannot be abstracted. Surprisingly, this benefit holds not only for passing properties, but also for *failing* properties indicating that the abstract counterexample was mappable to a concrete counterexample. However, the experiments demonstrate that LTSABS often entails more cumulative verification resources than LTS, given the number of refinements.

While discouraging, we note that these benchmarks are fairly small and simple to solve using existing techniques. Table II, which repeats this experimental setup, demonstrates that our LTSABS technique provides a more consistent overall benefit on more complicated industrial designs. *M.out* in further tables indicate memory overflow. Abstraction-guided liveness-to-safety algorithm outperforms all other algorithms in terms of number of solved designs and computation time, beating LTS by a factor of $3.17\times$ (18152 vs. 57616 seconds). LTS and **k-liv** solve 17 and 5 designs, respectively. **BDD** was ineffective on these larger designs.

B. Transformation-Enhanced Liveness Verification

While LTSABS solved all the benchmarks provided in Table II, runtime tends to be quite large for some of them. We thus repeated this experiment after running the following sequence (decision is made based on our prior experiences) of transformation engines instead of BRN alone (as with the prior experiments): BRN,MOD,BRN,EQV,BRN,CUT,BRN,RET,BRN,CUT,BRN. These results are presented in Table III. Column 2 shows the design size in terms of number of registers after this transformation sequence, one of the observations is that an average 73% reduction in register count is achieved through these simplifications. Unlike column 9 in Table II, column 8 in Table III represent the total register count after including

Design	Regs	Result	LTS		LTSABS					k-liv	BDD
			Time (s)	Algo	Time (s)	Algo	#Refn	ShRegs	%Red.		
cuom2	29	pass	66	IC3	136	IC3	7	17	21	T.out	83
lmcs06prodcell3	161	pass	480	IC3	667	IC3	10	86	23	T.out	T.out
lmcs06prodcell4	161	pass	876	IC3	499	IC3	10	91	22	5926	T.out
lmcs06prodcell5	161	pass	2337	IC3	2164	IC3	11	77	26	T.out	T.out
lmcs06prodcell9	161	fail	18	BMC	84	BMC	5	25	42	T.out	T.out
lmcs06bc57sp6	116	fail	105	IC3	709	IC3	12	81	15	1796	274
lmcs06brp3	87	fail	69	BMC	414	BMC	7	62	14	2226	4744
arbixs08p03	26	pass	T.out	-	T.out	-	1	26	0	T.out	37
arbi0s32bugp03	130	fail	18	BMC	18	BMC	1	130	0	T.out	T.out
cujc128	129	pass	T.out	-	T.out	-	8	129	0	T.out	T.out
cujc32	33	pass	347	LOC	593	LOC	17	33	0	T.out	83

TABLE I. LIVENESS EXPERIMENTS ON HWMCC 2012 BENCHMARKS

Design	Regs	Result	LTS		LTSABS				k-liv	BDD
			Time (s)	Algo	Time (s)	Algo	#Refn	ShRegs		
D10	939	pass	10309	IC3	545	IC3	10	40	T.out	T.out
D11	18020	fail	35	BMC	207	BMC	19	521	T.out	T.out
D12	3698	pass	16	ITP	13	IC3	1	4	52	T.out
D13	5741	pass	28	ITP	31	ITP	1	10	324	T.out
D14	1566	fail	15	BMC	88	BMC	10	112	9219	T.out
D17	1737	pass	21	ITP	13	ITP	1	4	116	M.out
D20	8653	pass	T.out	-	420	LOC	2	49	T.out	T.out
D21	24749	fail	85	BMC	163	BMC	15	204	T.out	T.out
ICB1	683	pass	7343	IC3	4717	IC3	4	332	T.out	T.out
ICB2	683	pass	9170	IC3	4637	IC3	4	277	T.out	T.out
ICB3	683	pass	9689	IC3	1688	IC3	4	160	T.out	T.out
ICB4	683	pass	8861	IC3	5101	IC3	4	275	T.out	T.out
IE0	5631	pass	28	ITP	28	ITP	3	64	T.out	T.out
IE1	5631	pass	36	ITP	27	ITP	3	84	T.out	T.out
IE7	5631	pass	30	ITP	27	ITP	3	66	3259	T.out
PNG1	6315	pass	198	ITP	207	ITP	1	1	T.out	T.out
PNG2	6315	pass	214	ITP	184	ITP	1	1	T.out	T.out
GE	26672	pass	738	ITP	56	IC3	1	1	T.out	T.out

TABLE II. LIVENESS EXPERIMENTS ON INDUSTRIAL DESIGNS

the shadow registers. For simplicity sake we did not include the extra logic we construct for fairness in these numbers. LTSABS again outperforms all other techniques, solving all 18 of these benchmarks. LTS, k-liv, and BDD solve 18, 3, and 0 benchmarks, respectively. In comparison with LTSABS results in Table II, there is a cumulative 2.7× speedup with LTSABS *after* transformations (18152 vs. 6611 seconds), and none took more than 16 minutes to solve. Additionally note that significantly fewer refinements are necessary after the transformations. LTSABS is 1.74× faster than LTS in these experiments.

An additional experiment is detailed in the last four columns of Table III: the result of a precise LTS conversion followed by a highly-tuned localization-refinement engine. In theory, this approach is able to provide the same reduction benefits as LTSABS, since any of the shadow registers may be cutpointed. Similar to *%Red.* in Table I, columns 9 and 15 represent the percentage reduction achieved when we compare with the size of precise LTS (2× #regs). It is observed that the nature of the conversion logic tends to confuse localization-refinement heuristics into including much logic that is not truly unnecessary, causing inferior runtime and fewer solved problems vs. LTSABS.

C. Simulation-based liveness checking

Table IV shows experiments to demonstrate the scalability of our simulation-based liveness checking results on various designs with arrays. The second column lists the number of registers in the design, and column 3 lists the number of cells in all arrays in the design: #rows × #columns.

A runtime comparison is made between our simulation-based liveness checking algorithm (column 5) versus simulation on a bit-blasted (via ARY) then liveness-to-safety converted design (column 7) for 100000 simulation cycles. We have used single pattern random simulation in this set of experiments. The sixth column represents number of registers after ARY and LTS. In several cases (cells with *M.out*), simulation of the bit-blasted design exhausts memory before the transformed design behavior can be explored for that many simulation cycles. The designs with *unsolved* status indicate that complete simulation performed without finding any counterexample.

VI. CONCLUSIONS

While liveness and safety verification are in the same complexity class, in practice liveness verification tends to be significantly more complicated, in many cases requiring dedicated verification algorithms. Little prior work has addressed automated liveness verification on problems with memory arrays, compounding this problem in mandating bit-blasting in cases.

In this paper, we have presented three techniques to address these challenges. First, we present a technique to enhance a

Design	Regs	LTS		LTSABS					k-liv	BDD	LTS-LOC			
		Time (s)	Algo	Time (s)	Algo	#Refn	#Regs	%Red.			Time (s)	Algo	#Regs	%Red.
D10	152	126	IC3	217	IC3	2	297	2	T.out	T.out	991	IC3	303	0
D11	3464	304	LOC	227	BMC	3	3570	48	T.out	T.out	243	LOC	6900	0
D12	458	41	BRN	41	BRN	1	463	49	27	T.out	40	BRN	5	99
D13	1008	188	LOC	158	ITP	1	1020	49	229	T.out	199	ITP	22	99
D14	142	36	BMC	263	BMC	14	214	25	T.out	T.out	182	BMC	287	0
D17	491	45	BMC	52	IC3	1	492	50	36	T.out	45	BRN	11	99
D20	8031	1738	IC3	729	IC3	1	8078	50	T.out	T.out	T.out	-	668	-
D21	785	311	BMC	398	BMC	16	908	42	T.out	T.out	268	LOC	1509	4
ICB1	146	2056	IC3	424	IC3	7	188	36	T.out	T.out	673	IC3	291	0
ICB2	146	1013	IC3	278	IC3	9	179	39	T.out	T.out	787	IC3	290	1
ICB3	146	1090	IC3	404	IC3	6	239	18	T.out	T.out	1108	IC3	293	0
ICB4	146	944	IC3	374	IC3	8	211	28	T.out	T.out	973	IC3	293	0
IE0	1275	237	IC3	165	IC3	2	1313	49	T.out	T.out	189	IC3	473	81
IE1	1273	249	ITP	168	IC3	2	1303	49	T.out	T.out	204	ITP	736	71
IE7	1275	244	ITP	170	IC3	2	1313	49	T.out	T.out	238	IC3	173	93
PNG1	3210	979	ITP	902	ITP	1	3212	50	T.out	T.out	T.out	-	5	-
PNG2	3208	978	ITP	888	ITP	1	3210	50	T.out	T.out	T.out	-	5	-
GE	8284	957	ITP	753	IC3	1	8329	50	T.out	T.out	864	ITP	465	97

TABLE III. LIVENESS EXPERIMENTS ON INDUSTRIAL DESIGNS AFTER AGGRESSIVE TRANSFORMATIONS

Design	Regs	Array cells	Result	SIM Liv (s)	ARY,LTS,SIM	
					#regs	Time(s)
unreach_case	11	2048	unsolved	93	4125	74
reg_plus_array_hit	1	2048	fail	7	4107	78
mem_16_by_16	29	1048576	unsolved	222	2097218	M.out
mem_8_by_8	21	2048	fail	4	4138	80
mem_8_8_asy_rd_syn_wr	19	2048	fail	4	4136	76
mem_16_16_asy_rd_syn_wr	27	1048576	fail	12	2097208	M.out
mem_16_16_asy_rd_asyn_wr	26	1048576	fail	34	2097206	M.out
mem_8_8_dl_syn_rd_syn_wr	29	2048	fail	4	4156	114
simple_array_livelock	5	8	fail	5	28	4

TABLE IV. SIMULATION-BASED LIVENESS CHECKING ON DESIGNS WITH ARRAYS

logic simulator to support native liveness checking, including designs with both registers and arrays. While the focus of this paper is more on formal vs. informal verification, we note that simulators are critical in a variety of proof and falsification frameworks, including semi-formal bug hunting and techniques to postulate or refute candidate invariants, design redundancies, etc. Bit-blasting arrays to registers is practically often too capacity-limiting.

Second, we explore the utility of abstraction of shadow registers during liveness-to-safety conversion. For complex problems, our experiments confirm a significant improvement due to this technique in terms of problem size, verification resources, and ability to yield a conclusive verification result. We additionally demonstrate the superiority of this technique over existing localization-refinement approaches. Finally, we demonstrate the importance of transformation-based verification in enhancing the scalability of proof and falsification techniques.

ACKNOWLEDGMENT

We would like to thank whole design and verification teams in IBM, who contributed industrial strength liveness instances with us.

REFERENCES

[1] Hardware model checking competition 2012. http://fmv.jku.at/hwmcc12.

[2] J. Baumgartner and H. Mony. Maximal input reduction of sequential netlists via synergistic reparameterization and localization strategies. In *CHARME*, 2005.

[3] J. Baumgartner and H. Mony. Scalable liveness checking via property-preserving transformations. In *DATE*, 2009.

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

[5] Aaron Bradley. Understanding IC3. In *SAT*. 2012.

[6] G. Cabodi, P. Camurati, L. Garcia, M. Murciano, S. Nocco, and S. Quer. Speeding up model checking by exploiting explicit and hidden verification constraints. In *DATE*, Apr. 2009.

[7] K. Claessen and N. Sörensson. A liveness checking algorithm that counts. In *FMCAD*, 2012.

[8] A. Kuehlmann and J. Baumgartner. Transformation-based verification using generalized retiming. In *CAV*, 2001.

[9] A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai. Robust boolean reasoning for equivalence checking and functional property verification. *TCAD*, 21(12), 2002.

[10] A. Mishchenko, S. Chatterjee, and R. K. Brayton. DAG-aware AIG rewriting: a fresh look at combinational logic synthesis. In *DAC*, 2006.

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

[12] K. Ravi, R. Bloem, and F. Somenzi. A comparative study of symbolic algorithms for the computation of fair cycles. In *FMCAD*, Nov. 2000.

[13] V. Schuppan. *Liveness Checking as Safety Checking to Find Shortest Counterexamples to Linear Time Properties*. PhD Thesis, ETH Zürich, 2006.

[14] V. Schuppan and A. Biere. Liveness checking as safety checking for infinite state spaces. *Electr. Notes Theor. Comput. Sci.*, 149(1), 2006.