Scalable Liveness Checking via Property-Preserving Transformations

Jason Baumgartner Hari Mony
IBM Systems & Technology Group, Austin, TX

Abstract

The ability of logic transformations to enhance safety property checking has been well-established, and many industrial-strength verification solutions accordingly rely upon a variety of synthesis and abstraction techniques for speed and scalability. However, little prior work has addressed the applicability of such transformations in the domain of liveness checking. In this paper, we provide the theoretical foundation to enable the efficient use of a variety of (possibly customized) transformations in a liveness-checking framework. We demonstrate the practical utility of this theory on a variety of complex verification problems.

1 Introduction

Formal verification refers to the process of exhaustively checking whether a design adheres to certain correctness properties. Properties may be categorized as either *safety* or *liveness* [1]. Safety properties are those which may be falsified by a finite-length counterexample scenario. For example, a property which checks that no *bad state* (e.g., *checkstop* condition) is ever reached would be a safety property.

Liveness properties may only be falsified by infinite-length counterexamples. For example, a property that a request will eventually get a grant would require an infinite-length counterexample to illustrate that the grant *never* occurs. Practically, such counterexamples are represented using a *lasso-shaped* finite-length trace with a prefix followed by a suffix loop of state transitions which may be indefinitely repeated to yield an infinite-length counterexample.

Traditionally, the verification of liveness properties requires dedicated and computationally expensive fixedpoint algorithms [2]. Recently, it has been demonstrated that liveness checking may be cast as safety checking with the overhead of approximately doubling the number of state elements of the design [3]. This overhead is due to the need to shadow previously-reached states, so that the state repetition comprising a lasso loop may be directly checked. The ability to perform liveness checking as safety checking has had a substantial impact on the scalability thereof, enabling a much richer set of verification and falsification algorithms to be applied than previously thought possible.

In addition to verification and falsification algorithms, a variety of *transformation* algorithms have been proposed to enhance the speed and scalability of safety property checking. The concept of using a series of synergistic logic transformations to iteratively reduce the size of a design under verification until it becomes tractable for terminal verification engines was proposed in [4], and since has been adopted by a variety of industrial-strength tools such as those of IBM [5] and Synopsys [6]. This paradigm has also been adopted by ABC [7], which recently has won the 2008 Hardware Model Checking Competition in large part due to the integration of a synergistic suite of transformations [8].

While the results of [3] imply that one may apply transformations to the safety-converted representation of a liveness problem, such an approach is inefficient for several reasons. First, the conversion of liveness to safety entails a substantial increase in logic size, which entails runtime overhead to the transformation algorithms. Second, and much more significantly, the nature of this conversion renders several useful transformations to be highly ineffective.

In this paper, we provide the theoretical foundation to allow a variety of (possibly customized) transformations to be applied directly to liveness-based testbenches. Aside from the consideration of specific types of shadow logic which may be suppressed [9], we are unaware of any prior work addressing the use of such transformations in a liveness-checking framework. Coupled with the results of [3], we feel that this work constitutes a straight-forward extension of industrial-strength safety property checkers into the domain of scalable liveness checking. We additionally note that these results provide insights into broader types of minimizations applicable directly to the state-repetition detection logic [3, 9].

2 Preliminaries

Definition 1. A *netlist* is a tuple $\langle\langle V, E\rangle, G, T, L, F\rangle$ comprising a finite directed graph with vertices V and edges $E \subseteq V \times V$. Function $G : V \mapsto types$ represents a mapping from vertices to $gate\ types$, including primary inputs,

978-3-9810801-5-5/DATE09 © 2009 EDAA

registers, and combinational gates with various functions. The register is the only sequential gate type, which has a designated *initial value* (which specifies its value at time 0) as well as a *next-state function* (which defines its time $i+1$ behavior). Set $T \subseteq V$ represents the *safety properties*, and set $L \subseteq V$ represents the *liveness properties*. Set $F \subseteq V$ represents the *fairness constraints*.

Definition 2. A *trace* is a temporal sequence of valuations to netlist vertices which is consistent with G. These valuations may be *Boolean* or *undefined*.

The *verification goal* associated with a safety property gate is to obtain a counterexample trace illustrating an assertion of that gate, or to prove that no such trace exists. The verification goal associated with a liveness property gate is to obtain a counterexample trace illustrating that gate remaining asserted forever, which also asserts every fairness gate infinitely often, or to prove that no such trace exists.

Practically, it is necessary to represent traces over a finite duration, even for liveness counterexamples. This may be achieved by referencing a special *LOOP* gate which nondeterministically 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 netlist at the end of the loop matches the current state at the beginning of the loop. Note that the corresponding liveness gate must remain asserted during this trace, and that each fairness gate must assert at least once within the *LOOP* assertion suffix. As noted in [3], this counterexample representation may be directly synthesized to convert a liveness-based netlist N to a safety-based netlist N', using a shadow register r' for every original register r against which a state repetition may be checked, during which the behavior of the liveness and fairness gates may be validated as constituting a counterexample scenario.

Definition 3. A trace is an *adequate counterexample* for property p if simulating its input sequence, while populating arbitrary Boolean values in place of any undefined input valuations, constitutes a valid counterexample for p.

In a transformation-based verification framework [4], a reduced netlist N' may be created from an original netlist N, and verification be performed on N' in place of N. We refer to such a process as *sound* if any correctness proof obtained on N' implies a corresponding proof on N, and *complete* if any adequate counterexample trace q' obtained on N' may be mapped to an adequate counterexample q on N using a *trace-lifting* procedure.

The trace generalization of Definition 3 has several applications in a transformation-based verification framework. First, certain inputs may be undefined in a counterexample trace if they are irrelevant to the algorithm used to falsify a property, perhaps as a byproduct of a simplifying transformation from N to N'. This definition reflects a commonly-used technique of populating missing trace valuations into q through simulation. Second, transformations may also eliminate registers from N to N', and thus entail that the *LOOP* assertion illustrated in a liveness counterexample obtained over N' may not directly constitute a state to next-state repetition with respect to N. However, a liveness counterexample obtained over N' may nonetheless preserve adequacy when mapped to N, possibly through a customized transformation or trace-lifting procedure.

3 Cone-of-Influence Reduction

A *cone-of-influence reduction* consists of discarding all gates which are not in the fanin of the property or fairness gates. The trace-lifting process for this reduction consists merely of mapping trace valuations from gates of N' directly to the corresponding gates of N. In conjunction with other transformations such as redundancy removal, this technique is capable of dramatically reducing netlist size.

Theorem 1. Consider netlist N' obtained from N by performing a cone-of-influence reduction. Provided that the trace lifting process is customized to unfold the lasso loop until a state repetition is witnessed over N, verifying N' in place of N is sound and complete.

Proof. Because a cone-of-influence reduction cannot alter the behavior of any property or fairness gate, it clearly is *sound* [9]. To demonstrate *completeness*, we need to prove the adequacy of any counterexample q' obtained on N' with respect to the corresponding property of N. Let q be the trace obtained by simulating the mapping of q' to N, populating 0's in place of any undefined input valuations[1] – e.g., to inputs in N but not in N'.

We may view a cone-of-influence reduction as an acyclic partition of N into $\langle N', N'' \rangle$, where q and q' are equivalent with respect to corresponding gates of N'. While q' will illustrate a lasso state repetition at time-frames i and $i+j$, trace q may not illustrate a repeated state at these time-frames because registers may exist in N''. However, since netlists are finite, N must exhibit a repeated state in q possibly at later time-frames: the behavior of N'' under the lasso-shaped input stimulus of q will eventually witness a state repetition which coincides with the repeating input pattern after adequate unfoldings of the loop of q'. Thus through adequate unfolding of the lasso loop in the finite trace representation, the resulting q will become an adequate counterexample for the corresponding target of N. □

[1] This result holds as long as undefined input valuations are selected which preserve a lasso input behavior, even if not populated as all 0's. However, 0 valuations are often desirable in practice to avoid creating unnecessarily cluttered stimulus in a counterexample.

Theorem 1 indicates that while counterexample q' obtained upon reduced netlist N' may not directly constitute a valid counterexample lasso for the original netlist N, it nonetheless contains adequate information such that simulation of its input sequence on N will eventually yield a valid counterexample lasso. It was also demonstrated in [9] that cone-of-influence reduction preserves liveness checking; this result is included here since **(1)** it illustrates an efficient simulation-based trace reconstruction procedure to obtain an adequate lasso on N given an adequate trace on N' (not included in [9]), and **(2)** it illustrates the significant role of *trace adequacy*, allowing us to broadly categorize and support various transformations as is done in Section 4.

While a cone-of-influence reduction may clearly be performed *after* a translation to safety, note that the state repetition check will include every register (shadow and original) in the cone of influence after that translation, regardless of the situation before, as was also noted in [9]. Hence Theorem 1 may enable a substantial reduction in verification resources through an efficient simulation-based trace-lifting procedure to compute a valid *LOOP* assertion.

4 Trace-Equivalence Based Transformations

Definition 4. Netlists N and N' are termed *trace-equivalent with respect to bijectively-mapped gate set* V_1 *and* V_1' if any trace on N, when projected down to V_1, has a corresponding trace over V_1' in N' and vice-versa.

Theorem 2. Consider netlist N' derived from N through any transformation which preserves trace equivalence with respect to inputs, property gates and fairness gates. Verifying N' in place of N is sound and complete.

Proof. Consider any counterexample q' which violates property l' on N'. The trace equivalence of Definition 4 means that trace q obtained by resimulating the mapped input valuations of q' on N will demonstrate identical behavior to the corresponding property and fairness gates. Since q' is an adequate counterexample for l' on N', trace q may be converted into an adequate counterexample for the corresponding property l on N through lasso unfolding, as follows from Theorem 1. Thus establishes *completeness*. *Soundness* follows by noting that this trace equivalence preserves all valid counterexamples. □

Theorem 2 implies that a wide variety of transformations may safely be used to enhance liveness checking. For example, virtually all logic synthesis optimizations adhere to this criterion, including redundancy removal [10, 11], combinational rewriting [12, 13], and dependent register elimination [14]. Netlist transformation-based techniques for word-level reductions may also fall into this categorization [6]. In addition to reducing logic on their own, such transformations often enable substantial portions of the netlist to fall out of the cone of influence, enabling even greater property-preserving reductions as per Theorem 1.

Other transformations have been proposed which are more general than strict input / output preserving synthesis-style optimizations, yet which preserve trace-equivalence given an appropriate input-mapping function. For example, the input reparameterization technique of [15] identifies a cut $\langle N_1, N_2 \rangle$ of netlist N such that every gate in N_1 is combinationally sensitized by an input, and replaces N_1 by a logic component N_1' over a smaller set of fresh inputs such that N_2 of $N = \langle N_1, N_2 \rangle$ is trace-equivalent to N_2 of $N' = \langle N_1', N_2 \rangle$. To enable the use of this technique for safety property checking where N_2 includes the property gates, a satisfiability-based trace-lifting procedure is provided to map safety counterexamples from N' to valid counterexamples on N, identifying valuations to inputs of N_1 which cause it to produce the sequence of valuations seen on the cut boundary between N_1' and N_2 of N'. The following generalization of Theorem 2 illustrates that liveness checking is also preserved with such transformations.

Theorem 3. Consider netlist N' derived from N through any transformation which preserves trace-equivalence with respect to property and fairness gates (not necessarily inputs), which is accompanied by a trace-lifting function f' which maps trace q' of N' to q of N which produces identical behavior to property and fairness gates. Verifying N' in place of N is sound and complete.

Proof. The trace-equivalence of property and fairness gates from N to N' establishes *soundness*. *Completeness* follows as a straight-forward extension to Theorem 2 through the use of f' to obtain an adequate counterexample on N from any adequate counterexample on N'. □

Trace-equivalence preserving optimizations may be applied *after* a conversion to safety, noting that optimality potential (e.g., redundancy) over a pair of registers r_1 and r_2 likely entails comparable optimality potential over the corresponding shadow registers r_1' and r_2'. However, the logic growth entailed by the conversion to safety needlessly entails additional runtime of the optimization procedure. Furthermore, as noted in Section 3, the chance that logic will fall out of the cone of influence through such optimizations is substantially diminished if performed *after* a conversion to safety. However, in cases, additional reductions may be obtainable only after a conversion to safety, e.g., to eliminate redundancy between an original and shadow register if the former is constant after initialization [9].

5 Retiming

Retiming is a technique which reduces the number of registers in a netlist by shifting them across combinational

gates. The number of registers moved backward (faninwise) across a gate is referred to as its *lag*, correlating to the number of time-frames that its behavior has been delayed [16]. *Peripheral retiming* allows registers to be forward-retimed across inputs (*borrowed*) as well as property and fairness gates (*prefix-discarded*). *Normalized retiming*, where every gate has a negative finite lag and hence the retimed netlist can be guaranteed to have an equivalent initial state set as the original, has been proposed as a safety-preserving netlist transformation in [4]. When lifting a retimed trace, valuations from a retimed gate g' which was lagged by $-j$ are skewed forward by j time-frames before being applied as valuations to the original gate g. Missing prefix input valuations are derived from the initial values of retimed registers borrowed from those inputs [4].

Theorem 4. Consider netlist N' formed from N through normalized peripheral retiming, where no liveness gate was forward-retimed. Provided that any lifted adequate counterexample from N' has a valid lasso recalculated over N, verifying N' in place of N is sound and complete.

Proof. First assume that there exists a valid infinite-length counterexample q on N, and consider the corresponding retimed trace q' on N' formed by the inverse of the trace-lifting process mentioned above. While retiming may lag gates by differing amounts, normalized retiming may only eliminate prefix time-frames from N'. Thus q' would also illustrate the retimed liveness gate asserted forever. Furthermore, because fairness constraints must hold infinitely often, any prefix-discarding of these gates cannot prevent q' from being a valid counterexample. The *leading* (opposite to lagging) of normalized-retimed gates merely shifts their valuations within and across the boundaries of the lasso loop of q vs q'. Retiming thus preserves any valid counterexample which exists on N, hence is *sound*.

To establish *completeness*, we first note that lifting any adequate infinite-length counterexample q' on N' yields an adequate counterexample q on N. This follows since: **(1)** trace-lifting merely extends the lasso prefix and shifts valuations within its suffix conversely to the above reasoning, and **(2)** we disallow prefix-discarding of registers from retimed liveness gates,[2] ensuring that they remain asserted in the lasso prefix as well as the suffix. The need to recalculate the lasso loop in a finite representation of q is as follows.

The trace-lifting process for retiming entails that signals which were led more than others will have deeper valuations in the lifted trace. Unlike in safety checking where such deeper valuations are unimportant after the assertion of the safety property, in liveness checking all such valuations are likely important to preserve the lasso behavior. Thus while q' may illustrate a valid lasso on N', the fact that inputs may be lagged by different amounts ambiguates precisely where the lasso loop begins and ends in q: it is necessary to take lags into account on a per-input basis when deciding how to unfold the lasso loop. Since lags are not components of a trace, it is critical when computing q to perform a simulation procedure similar to Theorem 1, unfolding input valuations from the lasso of q' until a valid lasso loop for N is produced in q. □

There are several noteworthy points regarding Theorem 4. First, similarly to Theorem 3, trace adequacy from N' to N follows from an analysis of property and fairness gate behavior regardless of the fact that registers of N' may not correlate to registers of N. Second, the need for recomputation of the lasso loop during trace lifting is due to the fact that no common-lag restrictions are imposed on N', motivating Corollary 1.

Corollary 1. Consider netlist N' formed from N through normalized retiming, without peripheral retiming. Verifying N' in place of N is sound and complete.

Retiming is capable of yielding dramatic reductions in netlist size, particularly on high-performance pipelined circuitry; in practice, *peripheral retiming* enables a substantial percentage of these reductions [4]. Note that peripheral retiming subsumes the elimination of "transition input variables" which merely delay inputs which otherwise are not sampled [9]. While one could retime a safety-converted netlist, the nature of the safety conversion renders retiming to be highly ineffective. The *data-hold* paths around the shadow registers preclude them from being retiming-reduced whatsoever (without intricate clock-domain aware analysis). Additionally, the state-repetition detection logic somewhat diminishes the ability to retime-reduce the original registers, as leading the original registers by differing amounts entails the penalty of injecting retimed registers over the repetition detection logic. Thus retiming is much more effective as a pre-safety-conversion transformation.

6 State-Folding Abstraction

Transformations such as phase abstraction [17] unfold next-state functions modulo some constant c. As such, each transition of a state-folded netlist N' correlates to c transitions of an original netlist N. During this process, c copies of every original gate are created in N', correlating to the behavior of that original gate at different modulo-c time-frames. In other words, each copy g_j at time i correlates to the original gate g at time $c{\cdot}i{+}j$, for $j \in 0, \ldots, c{-}1$. Trace-lifting effectively consists of reversing this *gate, time* mapping. Coupled with light-weight redundancy removal and cone-of-influence reduction, this transformation may yield

[2] A comparable result is obtainable without this restriction by conjuncting the retimed liveness gate with the initial value of every register prefix-discarded across it.

dramatic reductions on *clocked* netlists where many registers toggle at most once every c consecutive time-frames. It is demonstrated in [17] that safety property checking is preserved through phase abstraction by disjuncting over each copy of the safety property gate.

Theorem 5. Consider netlist N' obtained from N by a modulo-c state-folding abstraction, where each liveness gate l' in N' is a *conjunction* of each of the c copies of the corresponding liveness gate l from N, and each fairness gate f' in N' is a *disjunction* of each of the c copies of the corresponding fairness gate f in N. Verifying N' in place of N is sound and complete.

Proof. Consider any counterexample q in N, whose loop is from time $i, \ldots, i+j$. We note that it is possible to view this counterexample as one whose loop start and end time are multiples of c by delaying its start by $\big((k \cdot c) - i\big) \bmod c$ time-frames for an adequately large k to ensure a nonnegative modulo argument, and by unrolling the loop until its length becomes a multiple of c. It is precisely this view of the counterexample which will be preserved though state-folding abstraction, as the conjunction over the abstracted liveness gate copies ensures that the corresponding original liveness gate remains asserted throughout the lasso, and the disjunction over the abstracted fairness gate copies ensures that the corresponding original fairness gates will each assert within its loop. As such, adequate counterexamples are preserved through trace lifting, hence this process is *complete*. *Soundness* follows though noting that all counterexamples of N are preserved though state-folding abstraction by manipulating their loop start and end time. □

Given a counterexample q on netlist N with a loop from time $i \cdot c$ to $(i+j) \cdot c$, a state-folded netlist N' will have a corresponding counterexample q' with a loop from time i to $i+j$. However, the potential need to delay the loop start and end time on N to align it with c and hence preserve its detectability on N' may entail that direct trace lifting yields a redundantly-long lasso in q. Practically, this is often a minor concern since c tends to be small (often 2 or 4), representing the period of an oscillating sub-circuit which in itself will preclude a redundantly-long lifted trace. Additionally, a simulation-based recomputation of the lasso loop of q may be performed to minimize its length if desired.

While one could perform phase abstraction after a conversion to safety, doing so is less likely to yield a beneficial reduction. Due to the state repetition check, many of the shadow registers are likely to remain in the cone of influence even though the corresponding original registers otherwise would not have, in turn retaining those corresponding original registers. Additionally, since the shadow registers are unclocked, they sample the original registers at every time-frame. It is thus even less likely that the original registers will become irrelevant through other transformations.

7 Localization

Localization is a sound yet incomplete transformation which through *cutpointing* replaces arbitrary gates by inputs. Coupled with a cone-of-influence reduction, this often enables dramatic netlist reductions. To cope with spurious counterexamples, *refinement* is often performed to eliminate those cutpoints which are deemed responsible for the spurious failures [18]. Localization may clearly be performed *after* a conversion to safety, eliminating some of the original and shadow registers. Additionally, localization may be performed *before* a conversion to safety, since doing so will yield a result obtainable by a post-conversion localization. Customized localization-refinement algorithms may also be used to directly subset the shadow registers, which is a sound yet incomplete transformation [9].

8 Experimental Results and Conclusion

In this section we provide experimental results on four industrial testbenches to illustrate the capability of our theory to reduce proof and falsification resources in liveness checking. L2_CIU checks arbitration validity for a bus interface unit. L2_DIR comprises logic which arbitrates among memory requests for a L2 cache. EMQ_SLB validates that a data address-segment translation unit properly arbitrates and routes requests among various sources. I_ERAT checks the proper processing of requests by an effective-to-real instruction-address translator.

Table 1 illustrates the best runtimes we obtained for these testbenches using the various techniques presented in this paper. Refer to [5] for a discussion of an automated algorithm-scheduling process; the only customization that we performed to this highly-tuned expert-system framework was biasing to delay the liveness-to-safety conversion until other transformations had exhausted their reduction potential. All experiments were run on a 2.1 GHz processor, using the IBM internal verification tool *SixthSense*.

To illustrate the weakness of state-folding abstraction after a conversion to safety, we analyzed L2_CIU and L2_DIR from Table 1. If we perform a **LIV** conversion to safety immediately after the **MOD** state folding transformation, we obtain netlists with 940 and 294 registers, respectively. If we inject **LIV** before this **MOD**, we end up with 1409 and 557 registers, respectively – an increase of 49.9% and 89.5%, respectively. To illustrate the weakness of retiming after a conversion to safety, we performed a comparable experiment. If we run **LIV** after the retiming transformation **RET**, we obtain 722 and 234 registers, respectively. If we inject **LIV** before this **RET**, we end up with 845 and 274 registers, respectively – an increase of 17.0% and 17.1%, respectively, noting that the reduction in Table 1 by the **RET** transformation is otherwise 22.2% and 27.8%, respectively.

L2_CIU	Initial	COI	COM	MOD: c=4	RET	COM	LIV	COM	LOC	CUT	RCH: 10
ANDs	15283	7323	4051	3956	5225	4785	9703	7329	1080	934	11 sec
Registers	3696	1497	513	467	364	363	728	728	68	68	147 MB
L2_DIR	Initial	COI	COM	MOD: c=4	RET	EQV	LIV	BMC: 20			
ANDs	8432	1810	1235	737	904	725	2515	21 sec			
Registers	1558	447	292	144	114	113	232	202 MB			
EMQ_SLB	Initial	COI	COM	MOD: c=2	RET	EQV	LIV	COM	BMC: 10		
ANDs	29721	14521	1986	820	921	811	4564	2274	40 sec		
Registers	6816	3514	502	268	205	202	412	412	99 MB		
I_ERAT	Initial	COI	COM	MOD: c=2	LIV	COM	LOC	LOC	RCH: 225		
ANDs	112497	101163	74976	71461	315600	156994	1703	632	17870 s		
Registers	18115	16272	12895	12215	24434	24434	204	111	2781 MB		

Table 1: Transformation-enhanced liveness checking scalability experiments. **COI**: initial cone-of-influence reduction; all subsequent columns include a post-transform cone-of-influence reduction. **COM**: a set of combinational rewriting algorithms similar to those of [7]. **CUT**: input reparameterization [15]. **EQV**: sequential redundancy removal [11]. **LIV**: liveness-to-safety conversion [3]. **LOC**: localization [19, 15]. **MOD**: modulo-c state folding with combinational redundancy removal [17]. **RET**: min-area peripheral retiming [4]. **BMC**: SAT-based bounded falsification. **RCH**: BDD-based reachability. Resources shown in the final "solving" column are cumulative across all algorithms. The corresponding number represents the time-frame (with respect to the transformed netlist) at which the solution was obtained.

In L2_DIR, the properties fail after 20 steps of bounded analysis on the transformed netlist. This correlates to 96 time-frame traces on the original netlist. In EMQ_SLB, the properties fail after 10 steps of bounded analysis on the transformed netlist, correlating to 30 time-frame traces on the original netlist. Without our ability to transform the pre-safety converted netlist, our best runtimes for transformations and **BMC** were 11.4× and 2.3× greater, respectively.

The impact of our techniques on passing properties is even more pronounced. In addition to the above-mentioned post-**LIV** weaknesses of **RET** and **MOD**, **LOC** is ineffective at compensating for this relative logic bloat and less effective at eliminating original logic, often resulting in an inconclusive verification result. On L2_CIU, without our ability to transform the pre-**LIV** netlist, our best reduction had 482 registers – too large for a feasible proof. On I_ERAT, we could not achieve an adequately small localization even within 24 hours: our best attempt yielded 11079 registers and still was prone to spurious counterexamples.

These results clearly illustrate a profound capability of tailored transformations to enhance the scalability of proof and falsification algorithms on liveness-based testbenches.

References

[1] L. Lamport, "Proving the correctness of multiprocess programs," in *IEEE Trans. Software Engineering*, 1997.

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

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

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

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

[6] P. Bjesse, "A practical approach to word level model checking of industrial netlists," in *CAV*, July 2008.

[7] Berkeley Logic and Synthesis Group, *ABC: A System for Sequential Synthesis and Verification*. http://www.eecs.berkeley.edu/ alanmi/abc.

[8] Hardware Model Checking Competition 2008. http://fmv.jku.at/hwmcc08.

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

[10] A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification," *TCAD*, vol. 21, Dec. 2002.

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

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

[13] M. Case, V. Kravets, A. Mishchenko, and R. Brayton, "Merging nodes under sequential observability," in *DAC*, June 2008.

[14] C.-C. Lee, J.-H. Jiang, C.-Y. Huang, and A. Mishchenko, "Scalable exploration of functional dependency by interpolation and incremental SAT solving," in *ICCAD*, Nov. 2007.

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

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

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

[18] P. Chauhan et al., "Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis," in *FMCAD*, Nov. 2002.

[19] K. L. McMillan and N. Amla, "Automatic abstraction without counterexamples," in *TACAS*, April 2004.