k-FAIR = k-LIVENESS + FAIR
Revisiting SAT-based Liveness Algorithms

Alexander Ivrii, Ziv Nevo, Jason Baumgartner IBM Corporation

***Abstract*—We revisit the two main SAT-based algorithms for checking liveness properties of finite-state transition systems: the *k*-LIVENESS algorithm of [1] and the FAIR algorithm of [2]. These approaches are fundamentally different. *k*-LIVENESS works by translating the liveness property together with fairness constraints to the form FGq, and then bounding the number of times the variable q can evaluate to false. FAIR works by finding an over-approximation R of reachable states, so that no state in R is contained on a fair cycle. Each technique has unique strengths on different problems. In this paper, we present a new algorithm *k*-FAIR that builds upon both techniques, synergistically leveraging their strengths. Experiments demonstrate that this combined approach is stronger than running both in parallel.**

I. Introduction

Efficient verification of liveness properties remains an important unsolved problem. A common approach is based on the *liveness-to-safety* translation [3] that converts liveness properties to safety properties, enabling the use of any safety checking technique. In practice this translation works very well especially for failing properties, though suffers from the severe performance penalty of doubling the number of state variables. More direct SAT-based approaches have thus been proposed: FAIR [2] and *k*-LIVENESS [1]. In this paper, we revisit these approaches, and present a new algorithm *k*-FAIR that combines the strengths of both in a way that outperforms running them in parallel.

A liveness property can be converted to form FGq, meaning that on every path variable q must eventually evaluate to true forever [1]. A counterexample would illustrate q evaluating to false infinitely often. As the state-space of hardware models is finite, such a counterexample may be represented as a lasso-shaped trace, consisting of a prefix from an initial state to a $\neg q$-state s, and a repeating loop suffix from s back to itself.

Given a property FGq, *k*-LIVENESS [1] attempts to bound the number of times that q can evaluate to false. Effectively, this technique checks a sequence of safety properties p_k which evaluate to false when q evaluates to false at least $k+1$ times. Initially $p_0 = q$, and p_{k+1} is obtained from p_k by adding "absorbing logic" that masks one occurrence of $\neg q$. If $\neg p_k$ is proven valid, FGq is clearly valid. A bounded counterexample to $\neg p_k$ does not guarantee the existence of a counterexample for higher bounds, though if it exhibits a repeated state sequence within which q evaluates to false, it is a valid unbounded counterexample. Given a finite state space, for suitably-large k, either $\neg p_k$ will be proven or will yield a valid unbounded counterexample. *k*-LIVENESS is thus sound and complete. As noted in [4], in practice unbounded counterexamples can often be detected even for small values of k. Given the close relation between models being checked for increasing k, an *incremental* model checker such as IC3 [5], [6] offers the advantage of reusing information such as bounded and absolute invariants between each query.

FAIR [2] is an iterative algorithm that incrementally learns information about reachable states and the SCC-closed regions of the state space. Roughly speaking, a *reachability assertion* R indicates that all the states on a potential lasso-shaped counterexample belong to R, while a *wall* W states that all states on the loop suffix of a potential counterexample either together belong to W or together belong to the complement of W. If one side of the wall W has no reachable states, the wall actually represents a constraint on all states on the loop of a potential counterexample, called *stabilizing constraints* in [1]. Specializing to liveness properties of form FGq, FAIR uses a SAT-solver to obtain a $\neg q$-state s, subject to the previously-discovered reachability assertions and walls. If this query is unsatisfiable, then FGq holds. Otherwise, FAIR tries to compute lasso-shaped counterexample for s, checking whether s is reachable from an initial state, then whether s can eventually transition to itself. If both queries are satisfiable, the liveness property fails. Otherwise, FAIR requires the underlying safety model checker to produce an inductive proof of unsatisfiability. If s is not reachable from an initial state, this proof represents a new reachability assertion. If s cannot transition to itself, this proof represents a new wall. [2] suggests several methods to discover new walls, including a method to generalize s to a set of states s_{gen}, so that no state in s_{gen} has a loop back to itself; equivalently, $\neg s_{gen}$ is a new stabilizing constraint. In either case, the algorithm makes progress and must eventually terminate with a conclusive verification result.

These two algorithms have different strengths. When FGq is valid, *k*-LIVENESS works well when a small value of k is sufficient to prove unsatisfiability; otherwise the underlying safety queries become unscalable as k becomes large. FAIR works well when inductive proofs restrict large portions of the search space; otherwise, too many iterations are required.

In this paper, we propose a new algorithm *k*-FAIR that combines ideas from both approaches. Similarly to *k*-LIVENESS, we pose a safety query that checks for a trace on which q evaluates to false at least k times. If unsatisfiable, the liveness property is proven. If satisfiable, we check whether the bounded counterexample has a repeated $\neg q$-state; if so, the liveness property is disproven. Otherwise, we select a $\neg q$-state s from the trace, and (similar to FAIR) check if it can eventually transition back to itself. If so, the liveness property

is disproven. Otherwise, we extract a new stabilizing constraint $c = \neg s_{gen}$, by generalizing s to a larger set of states s_{gen} without a self-loop. These stabilizing constraints are used to restrict every occurrence of $\neg q$ in future checks for a trace on which $c \rightarrow q$ evaluates to false at least k times. Note that when a new stabilizing constraint is discovered, there is no need to increase k for completeness, enabling convergence with smaller bounds than k-LIVENESS.

In Section II, we describe details for making these restrictions more efficient with IC3 queries, and for more-efficient detection of new stabilizing constraints. Originally [2] suggests to periodically look for *single-literal* stabilizing constraints. [1] improves upon this technique by considering all nets in a circuit as candidate constraints, and using the liveness signal q in a stronger way; however, this is purely a preprocessing technique. k-FAIR uses the best of both worlds: applying the method of [1] periodically, using the external reachability invariants and stabilizing constraints to strengthen the induction hypothesis.

II. k-FAIR

A. Algorithm Overview

Algorithm 1 k-FAIR

Input: Liveness property FGq
Data: Reachability invariants R, Stabilizing constraints S

```
1:  OnLoop ← CreateOnLoopReg ()
2:  r ← register with init = true and next = (OnLoop → q)
3:  p ← r, k ← 0, R ← ∅, S ← ∅
4:  while true do
5:      if (*) then
6:          (st, S) ← StabilizingConstraints(R, S)
7:          if (st = UNSAT) then
8:              return PASS
9:      (st, α, R) ← Run_kLIVENESS(p, R, S)
10:     if (st = UNSAT) then
11:         return PASS
12:     if α has a state repetition with ¬r then
13:         return FAIL
14:     if (*) then
15:         s ← Select last state on α with ¬r
16:         (st, β, S) ← Run_FAIR(s, R, S)
17:         if (st = SAT) then
18:             return FAIL
19:     if (*) then
20:         p ← AbsorbingLogic(p, r), k++
```

Our k-FAIR algorithm is depicted in Algorithm 1. It accepts a liveness property FGq (which embeds fairness constraints), and returns PASS or FAIL with counterexample. The algorithm incrementally updates two important structures: *reachability invariants* R (that constrain all states on a potential lasso-shaped counterexample), and *stabilizing constraints* S (that constrain all states on the loop of a potential lasso-shaped counterexample). In practice, each constraint in R and S is a clause (disjunction) over registers and internal nets.

Lines 1–3. Function `CreateOnLoopReg` creates a new register $OnLoop$ that is initialized to 0, which nondeterministically changes its value to 1 after which it remains 1 forever. We create a new register r with next-state function $OnLoop \rightarrow q$. It is easy to see that the validity of FGq is equivalent to the validity of FGr, and a counterexample to FGr is a counterexample to FGq. Intuitively, $OnLoop$ allows to efficiently pass information to the underlying safety model checker, while register r simplifies implementation details. Variable p represents the value of the current safety property. For clarity, index k corresponds to the safety property p_k.

Lines 5–8. Algorithm `StabilizingConstraints` derives new stabilizing constraints, accepting R and S and updating S. This function is similar to [1], except that it additionally uses R and S to restrict both current- and the next-states in the SAT-solver. Additionally, we have found it useful to reason about the original fairness constraints instead of q when looking for nets that stabilize to a constant value. Theoretically, this allows `StabilizingConstraints` to discover more stabilizing constraints as the sets R and S are extended elsewhere, justifying the value of running this function periodically. In cases, these new stabilizing constraints exclude *all* reachable states, in which case the algorithm terminates with PASS.

Lines 9–11. Function `Run_kLIVENESS`(p, R, S) checks whether an initial state can reach a $\neg p$-state, subject to constraints $R \wedge (OnLoop \rightarrow S)$. Equivalently, this checks for a path from an initial state, on which $OnLoop \wedge \neg q$ occurs at least k times under these constraints. In particular, the stabilizing constraints S must hold on every state after the first occurrence of $OnLoop \wedge \neg q$. This is slightly stronger than suggested in [1], where the stabilizing constraints are only used to restrict the $\neg q$-states. This function returns the verification status $st \in \{\texttt{SAT}, \texttt{UNSAT}\}$, counterexample α for $st = \texttt{SAT}$, and additional reachability invariants R discovered in the process. As in [1], we use an incremental IC3-engine, which reuses bounded and absolute invariants between runs; this allows to neglect explicitly passing R to this engine. Instead of synthesizing $(OnLoop \rightarrow S)$ using new logic, we have extended the IC3-engine to accept *clausal constraints* over registers and internal nets. In particular, for each clause $c \in S$, we pass the clausal constraint $\neg OnLoop \vee c$. If the verification status st returned by `Run_kLIVENESS` is `UNSAT`, the algorithm terminates with PASS.

Lines 12–13. If the safety query returns `SAT`, then as suggested in [4] we analyze the counterexample α to check if it exhibits a state repetition on which r evaluates to false. If so, the counterexample is valid and the algorithm terminates with FAIL. Additionally, we may manipulate α using the trace manipulation techniques described in [4] to improve the likelihood of producing a valid counterexample from α.

Lines 14–18. First, we select a $\neg r$-state s from α; by construction there are at least $k+1$ such states. In practice, the last such state is most effective, though any (or multiple) may

be selected. Function Run_FAIR(s, R, S) checks for a path from s back to itself, subject to constraints $R \wedge S$. If the result is SAT, a valid counterexample β exists and the algorithm terminates with FAIL. (A valid counterexample to FGr can be constructed by concatenating α and β). If $st =$ UNSAT, we use the technique of [2] to generalize s to a larger set of states s_{gen}, none of which can transition to s_{gen}. In this case, we update S by adding a new stabilizing constraint $\neg s_{gen}$. As in [2], we use an IC3-engine, which produces inductive invariants. To avoid trivial 0-length paths, we introduce an additional register Z with initial value 0 and next-state function 1, and the actual query checks whether $s \wedge \neg Z$ can reach $s \wedge Z$. Note that passing sets R and S to IC3 is not required for correctness, so using them most efficiently in the underlying IC3-engine poses a complex implementation choice. In our experience, having many redundant clausal constraints may slow down IC3 (hurting ability to reduce proof obligations by ternary simulation or alternative techniques). In our implementation, we pass S as clausal constraints and R as *clausal invariants*, the difference being that clausal invariants are ignored when reducing proof obligations.

Lines 19–20. If Run_FAIR was executed, a new stabilizing constraint was detected hence the algorithm made progress. Contrary to k-LIVENESS, adding absorbing logic is not required for completeness: we may continue with the same value of k (or even reduced k). In fact, FAIR can be seen as an instance of this algorithm when k is always zero.

B. Comparison to k-LIVENESS and to FAIR

k-FAIR effectively combines the strengths of k-LIVENESS and FAIR. If Run_FAIR is never executed (if the **if**-condition on line 14 is always false), then k-FAIR closely corresponds to k-LIVENESS modulo the ability to detect new stabilizing constraints via reachability invariants from IC3. k-FAIR can be viewed as k-LIVENESS extended with an additional technique to look for unbounded counterexamples. On the other hand, if AbsorbingLogic is never executed (if the **if**-condition on line 19 is always false), k-FAIR corresponds to an alternative implementation of FAIR. Though instead of using a SAT-solver to find candidate $\neg q$-states s and checking if they are reachable from an initial state, we search for such reachable states directly. Additionally, when s cannot reach itself, we only borrow the method of [2] that discovers stabilizing constraint and not a more general wall constraint. Arguably, this makes our implementation simpler, but may lose some potential power enabled by more general constraints.

III. Experiments

In this section we present our experimental results. The techniques described in this paper are implemented in the IBM formal verification tool *Rulebase: Sixthsense Edition* [7]. All experiments are executed on a 2.00 GHz Linux machine with an Intel Xeon E7540 processor, 16GB of RAM, and 3 hours time-limit. We used all single-property liveness benchmarks from the 2011–2017 Hardware Model Checking Competitions [8], as well as various proprietary industrial testcases. For a more realistic setup, the benchmarks are preprocessed using standard logic synthesis techniques (similar to ABC [9] commands *rewrite*, *lcorr* and *ssw*).

TABLE I
Summary of Experimental Results

	PASS solved	PASS time	FAIL solved	FAIL time
***k*-FAIR-fair**	108	338,475	89	351,634
***k*-FAIR-b50**	111	301,592	94	321,260
***k*-FAIR-b5**	122	166,655	104	240,458
***k*-FAIR-klive**	123	173,077	97	245,543
***k*-FAIR-klive-pre**	117	250,431	100	225,971
LTS-BMC	-	-	114	94,103
LTS-IC3	116	225,059	99	226,321
VBS	131	37,270	117	29,315
VBS without **klive**	130	43,661	117	29,349
VBS without **b5,b50**	131	37,510	116	31,326
***k*-FAIR-fair** & ***k*-FAIR-klive-pre**	124	175,581	107	154,646
LTS-IC3 & ***k*-FAIR-klive**	131	37,270	100	206,171
LTS-BMC & ***k*-FAIR-b5**	122	166,655	117	57,482

TABLE II
Proven k value for commonly-solved benchmarks

***k*-FAIR-**	**fair**	**b50**	**b5**	**klive**	**klive-pre**
average k	0	0.50	1.84	6.02	10.09

A. Review of results

The configurations evaluated include: The first five configurations are different variants of Algorithm 1. In ***k*-FAIR-fair**, Run_FAIR runs on every iteration of the main loop but the counter k is never incremented: this is "pure FAIR" mode. In ***k*-FAIR-b50** and ***k*-FAIR-b5**, Run_FAIR runs on every iteration of the loop, while the counter k is incremented on, respectively, every 50th and 5th iteration. In ***k*-FAIR-klive** and ***k*-FAIR-klive-pre**, Run_FAIR never runs, and the counter is incremented on every iteration of the loop: this is the "pure k-LIVENESS" mode. In all five variants, StabilizingConstraints runs on the first iteration of the main loop as preprocessing. Additionally, in the first four variants, StabilizingConstraints runs periodically (either each time the counter increments, or on every 50th iteration of the loop, whichever happens first). In ***k*-FAIR-klive-pre**, StabilizingConstraints does not run again, corresponding most closely to [1]. The last two configurations **LTS-BMC** and **LTS-IC3** correspond to the liveness-to-safety translation, followed by BMC (Bounded Model Checking) [10] and IC3, respectively.

Table I summarizes the experiments. Columns "PASS solved" and "FAIL solved" show the number of passing and failing instances, respectively, solved by a specific configuration. Columns "PASS time" and "FAIL time" represent the cumulative time in seconds for passing and failing properties, respectively. Benchmarks solved by preprocessing alone, and

TABLE III
COMPARISON OF "PURE FAIR" IN k-FAIR AND FAIR IN *IImc*

	PASS solved	PASS time	FAIL solved	FAIL time
***k*-FAIR-fair**	108 (18)	208,875	89 (21)	70,834
IImc-fair	101 (11)	277,657	70 (2)	242,762

those not solved by any configuration, are excluded from further consideration, leaving a total of 131 passing and 117 failing testcases. As BMC cannot prove properties, results are shown only for failing properties. Row "**VBS**" corresponds to the *virtual best* of all configurations. The last five rows represent selected portfolios of the configurations above. For example, row "**VBS** without **klive**" corresponds to running in parallel all configurations except **klive**. Row "***k*-FAIR-fair** & ***k*-FAIR-klive-pre**" corresponds to running in parallel the two configurations ***k*-FAIR-fair** and ***k*-FAIR-klive-pre**.

B. Overall summary

Simple liveness-to-safety followed by BMC is a very strong falsification strategy, solving all but 3 failing testcases. Interestingly, these 3 are solved by ***k*-FAIR-b5** (with one unique solve), and in each case the counterexample is detected by `Run_FAIR` vs. the state repetition check. This may be because candidate states returned by `Run_kLIVENESS` for large k have a higher chance to belong to a valid counterexample. The best two-engine parallel portfolio consists of **LTS-BMC** and ***k*-FAIR-b5**, solving *all* failing properties with a runtime improvement of 1.6 vs. **LTS-BMC** alone. A parallel portfolio running all seven configurations improves total runtime by an additional factor of 1.9.

For passing properties, the "pure k-LIVENESS approach with an incremental detection of stabilizing constraints" performs best (yielding one unique solve), outperforming both the "pure FAIR" approach, the liveness-to-safety followed by IC3, and the "standard k-LIVENESS approach" ***k*-FAIR-klive-pre**. A best two-engine portfolio consists of **LTS-IC3** and ***k*-FAIR-klive**, solving all passing properties in the smallest possible time.

C. Examining k sufficient for proof

There are 100 passing testcases (out of 131) solved by all five variants of k-FAIR. In Table II we restrict to these testcases and report the values of k sufficient for proof, averaged over all the testcases. Not surprisingly, this value is 0 in "pure FAIR" mode, and gradually increases to 6.02 as the variant changes to "pure k-LIVENESS." This table shows that stabilizing constraints based upon `Run_FAIR` reduce the value of k needed to obtain a proof. Without the incremental detection of stabilizing constraints based on `StabilizingConstraints`, the sufficient value of k is even larger.

D. Comparing *k*-FAIR-fair to IImc-fair

As an additional experiment, we compare ***k*-FAIR-fair** – our "pure FAIR" approach, and **IImc-fair** – the original FAIR algorithm of [2]. **IImc-fair** uses the implementation in *IImc* [11] with command *iimc -t fair -v1 --fair_timeout 10800*. The results are summarized in Table III. As before, we present data only for testcases solved by at least one configuration. The numbers in parentheses represent unique solves. Overall ***k*-FAIR-fair** performs substantially better than **IImc-fair**, both on passing and failing properties, though both variants have unique value. Unfortunately, a detailed comparison is difficult, as the two techniques are implemented in very different verification frameworks, and the improvements may be due to a large number of different factors, including an improved method in [1] to find stabilizing constraints, only looking for loops from a priori reachable states, and the syntactic check for a state repetition (for failing properties). In any case *the adaptation of FAIR presented in this paper seems as a viable alternative to the implementation in IImc [11].*

IV. CONCLUSION AND FURTHER WORK

In this paper we presented the algorithm k-FAIR, which combines the strengths of the prominent SAT-based algorithms for liveness verification: k-LIVENESS and FAIR. We experimented with several variants of k-FAIR and demonstrated that a combined portfolio approach brings unique value.

Fine-tuning the algorithm is likely to offer additional performance improvements. Each of the main methods `StabilizingConstraints`, `Run_kLIVENESS` or `Run_FAIR` may be the key to success, but may also be the bottleneck of the approach. Carefully balancing the effort spent on each component (e.g., by suitably imposing resource limits, or by increasing or decreasing k more aggressively) is a subject of further research. Another promising direction consists of tuning the underlying IC3-engine towards the safety queries posed by the algorithm. For example, one could attempt to leverage the fact that all safety queries made by `Run_kLIVENESS` (except for possibly the very last one) are satisfiable, while all safety queries made by `Run_FAIR` (except for possibly the very last one) are unsatisfiable. Additionally, one could attempt to devise better methods to pass constraints, invariants, etc. to the IC3-engine, and to use these more efficiently in the IC3-engine itself.

REFERENCES

[1] K. Claessen and N. Sörensson, "A liveness checking algorithm that counts," in *FMCAD*, 2012.

[2] A. R. Bradley, F. Somenzi, Z. Hassan, and Y. Zhang, "An incremental approach to model checking progress properties," in *FMCAD*, 2011.

[3] A. Biere, C. Artho, and V. Schuppan, "Liveness checking as safety checking," *Electr. Notes Theor. Comput. Sci.*, vol. 66, no. 2, 2002.

[4] G. Aleksandrowicz, J. Baumgartner, A. Ivrii, and Z. Nevo, "Generalized counterexamples to liveness properties," in *FMCAD*, 2013.

[5] A. Bradley, "SAT-based model checking without unrolling," in *VMCAI*, Jan. 2011.

[6] N. Eén, A. Mishchenko, and R. K. Brayton, "Efficient implementation of property directed reachability," in *FMCAD*, 2011.

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

[8] Hardware Model Checking Competition 2017. http://fmv.jku.at/hwmcc17.

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

[10] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, “Symbolic model checking without BDDs,” in *TACAS*, March 1999.
[11] A. R. Bradley and F. Somenzi and Z. Hassan, *IImc: an Incremental Inductive model checker*. https://github.com/mgudemann/iimc.