Avoiding the Shoals - A New Approach to Liveness Checking ⋆

Yechuan Xia[1], Alessandro Cimatti[2], Alberto Griggio[2], and Jianwen Li[1]

[1] East China Normal University, Shanghai, China
jwli@sei.ecnu.edu.cn
[2] Fondazione Bruno Kessler, Trento, Italy
{cimatti,griggio}@fbk.eu

Abstract. We present rlive, a new SAT-based model-checking algorithm for the verification of liveness properties of finite-state symbolic transition systems. Like other recent approaches, rlive works by reducing liveness checking to a sequence of safety checks. Similarly to FAIR, it incrementally strengthens the input system using constraints obtained by refuting candidate counterexamples to the input liveness property, assumed (w.l.o.g.) to be of the form FGq. Differently from FAIR (and crucially), however, instead of directly searching for lasso-shaped counterexamples visiting $\neg q$ infinitely-often, rlive searches for counterexamples incrementally, via a recursive chain of safety checks, each of which tries to determine whether it is possible to reach a $\neg q$-state from a given $\neg q$-state (which was previously determined to be reachable), in a manner similar to k-Liveness. When the current candidate counterexample is refuted, rlive exploits the inductive invariants generated by the (recursive) safety checks to restrict the search space, until either no more reachable $\neg q$-states remain, or a real lasso-shaped counterexample is found.
In this paper, we describe rlive in detail, prove its soundness and completeness, and compare it against the state of the art both theoretically and empirically. Our experimental results show that our implementation of rlive outperforms state-of-the-art implementations of FAIR, k-Liveness and other SAT-based liveness checking algorithms on a wide range of benchmarks from the literature.

1 Introduction

The design of efficient algorithms for model checking has been a major research challenge for over three decades. Following the SAT breakthrough in the late 90s [22,25], many novel SAT-based techniques have been proposed, which have tremendously increased the efficiency and scalability of (symbolic) model checking and its applicability to real-world systems (e.g., [6,24,21,20,8,15,27,17,18]). Although the vast majority of such approaches have focused on safety properties, their benefits have extended also to liveness model checking, thanks to the development of liveness verification algorithms that work by exploiting efficient

⋆ Jianwen Li is the corresponding author.

safety checkers, either via a monolithic reduction from liveness to safety [4], or via more sophisticated strategies that use safety checkers incrementally [13], exploiting also the inductive invariants generated when the verification is successful [9,16].

In this paper, we present a novel SAT-based liveness checking algorithm, which we call rlive, that also takes advantage of efficient safety model checkers and their capability of producing inductive invariants for verified properties. Like all other SAT-based approaches to liveness checking, rlive works on properties of the form FGq, stating that q has to eventually stabilize to true in all traces of the system, relying on standard procedures (e.g., [14,12]) for transforming a model checking problem for an arbitrary LTL property into this form.

Similar to the FAIR algorithm of [9], rlive then proceeds by refuting candidate counterexamples to the property, i.e. traces in which $\neg q$ holds infinitely often, using a sequence of calls to a safety checker, and exploiting the inductive invariants generated by such safety checks to prune the set of reachable $\neg q$-states, until either a real (lasso-shaped) counterexample for FGq is found, or no $\neg q$-states are reachable, implying that the property holds. However, in contrast to FAIR, which directly searches for lasso-shaped traces where $\neg q$ holds in at least one state of the loop, rlive searches for counterexamples incrementally, via a recursive chain of safety checks, each of which tries to determine whether it is possible to reach a $\neg q$-state starting from the successors of a previously-reached $\neg q$-state, in a manner conceptually similar to k-Liveness [13]. If a $\neg q$-state is found for the second time during this recursive chain, a (lasso-shaped) counterexample witnessing the violation of FGq is constructed, and the algorithm terminates. Otherwise, eventually one of the recursive safety checks will generate an inductive invariant C proving that no other $\neg q$-state can be reached from (the successors of) a given $\neg q$-state s. rlive then uses C to derive constraints that exclude s from the reachable states of the system, forcing it to (recursively) consider a different $\neg q$-state to continue the current candidate counterexample trace. Specifically, C is used to strengthen the target states to reach, by asking the safety checker to ignore $\neg q$-states whose successors are all contained in C (since all such states in C cannot visit $\neg q$ infinitely-often); furthermore, C can be used also to strengthen the transition relation of the input system, since no state in C can be part of a counterexample. To give this intuition, we refer to states in C as *shoals*, as they represent regions of the state space that must be avoided in order to not "get stuck" in the search for a counterexample. Eventually, the shoals (recursively) produced will either exclude all $\neg q$-states, thus proving that the input property holds, or compel rlive to find a lasso-shaped counterexample for it.

Intuitively, rlive effectively identifies counterexamples by searching, in a depth-first manner, for traces that contain as many $\neg q$-states as possible. Performing the search incrementally, by a sequence of simple reachability checks, turns out to be computationally cheaper than searching directly for loops in practice. Moreover, whenever the current candidate counterexample trace cannot be completed, the shoals obtained from the safety checks can be used globally to strengthen

the transition system and reduce the search space that needs to be explored, thus accelerating the convergence of the algorithm.

We have implemented rlive on top of the nuXmv model checker [10] which has a mature, state-of-the-art IC3 implementation, and compared it against state-of-the-art implementations of other SAT-based liveness checking algorithms, including FAIR, k-Liveness, and their recent combination called k-FAIR [16]. Our experimental results, conducted on a wide range of benchmarks taken from recent hardware model checking competitions [1,2], demonstrate the strengths of our algorithm: rlive solves more benchmarks than any other competitor in the given resource bounds, and very often with significantly shorter time.

Paper structure. The rest of the paper is structured as follows. After the introduction of the necessary background in Section 2, we describe rlive in Section 3 and prove its soundness and correctness. We compare rlive with related work in Section 4, and experimentally evaluate its performance in Section 5. Finally, we conclude in Section 6 outlining also directions for future work.

2 Preliminaries

2.1 Boolean Satisfiability

A *literal* is a Boolean variable or its negation. If l is a literal, we denote its corresponding variable with $var(l)$. A *cube* (resp. *clause*) is a conjunction (resp. disjunction) of literals. The negation of a clause is a cube and vice versa. A formula in *Conjunctive Normal Form* (CNF) is a conjunction of clauses. For simplicity, we also treat a CNF formula ϕ as a set of clauses and make no difference between the formula and its set representation. Similarly, a cube or a clause c can be treated as a set of literals or a Boolean formula, depending on the context.

We say a CNF formula ϕ is satisfiable if there exists an assignment of its Boolean variables, called a *model*, that makes ϕ true; otherwise, ϕ is unsatisfiable. A SAT solver is a tool that can decide the satisfiability of a CNF formula ϕ. In addition to providing a yes/no answer, modern SAT solvers can also produce *models* for satisfiable formulas, and *unsatisfiable cores* (UC), i.e. a reason for unsatisfiability, for unsatisfiable ones. More precisely, in the following we shall assume to have a SAT solver that supports the following API (which is standard in state-of-the-art SAT solvers based on the CDCL algorithm [19]):

- IS-SAT($\phi, \mathcal{A}$) checks the satisfiability of ϕ under the given assumptions $\mathcal{A}$, which is a list of literals. This is logically equivalent to checking the satisfiability of $\phi \wedge \bigwedge \mathcal{A}$, but is typically more efficient;
- GET-UC() retrieves an UC of the assumption literals of the previous SAT call when the formula $\phi \wedge \bigwedge \mathcal{A}$ is unsatisfiable. That is, the result is a set $uc \subseteq \mathcal{A}$ such that $\phi \wedge \bigwedge uc$ is unsatisfiable;
- GET-MODEL() retrieves the model of the formula $\phi \wedge \bigwedge \mathcal{A}$ of the previous SAT call, if the formula is satisfiable.

2.2 Boolean Transition Systems

A Boolean transition system *Sys* is a tuple $\langle X, I, T\rangle$, where X is a set of variables, and I and T are formulae. The state space of *Sys* is the set of possible assignments to X. $I(X)$ is a Boolean formula corresponding to the set of initial states, and $T(X, X')$ is a Boolean formula representing the transition relation, where $X' = \{x' \mid x \in X\}$ represent the next state variables. In the following, we extend the prime notation to states and formulae in the natural way. The state s_2 is a successor of a state s_1 iff $s_1 \wedge s_2' \models T$, which is also denoted by $(s_1, s_2) \in T$. A *finite* path of length k is a finite state sequence $s_1, s_2, \ldots, s_k$, where $(s_i, s_{i+1}) \in T$ holds for $(1 \leq i \leq k-1)$. An *infinite* path is an infinite state sequence $s_1, s_2, \ldots$, where $(s_i, s_{i+1}) \in T$ holds for $i \geq 1$. The number of states is finite for any (Boolean) transition system. An infinite path is *lasso-shaped* if it can be presented as $\alpha \cdot \beta^\omega$, where α is the finite prefix, e.g. $s_1, s_2, \ldots, s_{l-1}$, and β is an infinitely-repeating suffix, e.g. $s_l, s_{l+2}, \ldots, s_k$. A state t is reachable from s in k steps if there is a path of length k from s to t. Let S be a set of states in *Sys*. We overload T and denote the set of successors of states in S as $T(S) = \{t \mid (s,t) \in T, s \in S\}$. Conversely, we define the set of predecessors of states in S as $T^{-1}(S) = \{s \mid (s,t) \in T, t \in S\}$. Recursively, we define $T^0(S) = S$ and $T^{i+1}(S) = T(T^i(S))$ where $i \geq 0$; the notation $T^{-i}(S)$ is defined analogously. In short, $T^i(S)$ denotes the states that are reachable from S in i steps, and $T^{-i}(S)$ denotes the states that can reach S in i steps.

2.3 Invariant Checking

Let a Boolean transition system $Sys = \langle X, I, T\rangle$ be given. A Boolean formula P over X is an invariant iff it holds in all the reachable states of Sys. An invariant checker either proves that P holds for any state reachable from an initial state in I, or disproves P by producing a *counterexample*. In the former case, we say that the property is *proven* in the system, while in the latter case, the property is *disproved*. A *counterexample* is a finite path from an initial state s to a state t violating P, i.e., $t \in \neg P$; such a state is also called a *bad* state.

Invariant checking, also referred to as safety checking, is reduced to symbolic reachability analysis. Reachability analysis can be performed in a forward or backward search. Forward search starts from initial states I and searches for bad states by computing $T^i(I)$ with increasing values of i, while backward search begins with states in $\neg P$ and searches for initial states by computing $T^{-i}(\neg P)$ with increasing values of i.

State-of-the-art safety checking algorithms utilize SAT techniques to explore the state space so as to improve the overall performance dramatically. Representative approaches include IC3/PDR [8,15], interpolation-based model checking [20], combinations of IC3 with interpolation [27] or k-induction [17], and (forward and backward) CAR [18]. In the following, we abstract from specific invariant checking algorithms, and assume that they implement the following API:

- CHECK-REACHABLE$(I, T, \neg P)$ denotes a generic procedure for safety checking. It takes as input a set of initial states I, the transition relation T, and the negation of the candidate invariant P. CHECK-REACHABLE returns *unsafe* if P is not an invariant. Otherwise, it returns *safe*.
- GET-INVARIANT() retrieves an inductive invariant proving that the bad states are unreachable, i.e. a set ι of states closed under T, containing the states reachable from I, and not intersecting $\neg P$. More formally, ι is such that $I \models \iota$, $\iota \wedge T \models \iota'$, and $\iota \models P$.
- GET-CEX-TRACE() retrieves, if the property is violated, the counterexample found by the safety checker, i.e. a finite path from I to $\neg P$.

2.4 Liveness Checking

We now consider the general model checking problem, denoted $Sys \models \phi$, where ϕ is a formula in Linear Temporal Logic (LTL) [23]. Following the standard automata-theoretic approach [26], the problem can be reduced to checking $Sys \times \mathcal{A}_{\neg\phi} \models FGq$, where $\neg q$ can be seen as the Büchi acceptance condition of $\mathcal{A}_{\neg\phi}$. (Symbolic techniques such as [14,12] can be used in practice to encode such reduction.) FGq intuitively means that, in any satisfying trace, q eventually holds in all the future states, so that the acceptance condition $\neg q$ can only be visited a finite number of times. Dually, a counterexample is an infinite path where $\neg q$ is visited an infinite number of times, i.e. a trace satisfying $GF\neg q$.

In the following, we focus on the (simplified) $Sys \models FGq$ problem, referred to as *liveness checking*. If the property is violated, there always exists a lasso-shaped counterexample[3], i.e., an infinite path $\alpha \cdot \beta^\omega$ where (i) the prefix α is a finite trace of Sys whose last state t violates q, i.e., $t \in \neg q$, and (ii) the infinitely-repeating suffix β is a path in Sys from a successor of t to t. We refer to a state $t \in \neg q$ as a $\neg q$-state.

The algorithms for liveness checking are more complicated than those for invariant checking. In order to show that a candidate invariant does not hold, it is sufficient to find a finite path. Liveness checking, on the other hand, requires finding an infinite (lasso-shaped) counter-example (or proving that none exists). The most effective solutions to liveness checking are based on invariant checking. The most relevant to our work are the following.

- The L2S [4] (Liveness-to-Safety) construction introduces a copy of the state variables in Sys, to record the first state of the loop, and a fresh variable $inLoop$, to record that the loop has started. The state vector copy is non-deterministically assigned a state violating q, i.e. the start of the loop, and can never change after that. The search tries to reach a state where each state variable has the same value as its copy and $inLoop = true$, which implies that a violating lasso is detected. This translation is sound and complete.

[3] Note that this fact only holds in the finite-state case; for infinite-state systems, the existence of a lasso-shaped counterexample is not guaranteed in case of violation.

Algorithm 1 k-FAIR = k-Liveness + FAIR

```
1: Liveness Property: FGq
2: procedure k-FAIR(I, T, q)
3:     k := 0, C := ∅, W := ∅                          // C: states not in loop, W: wall
4:     while true do
5:         if not IS-SAT(¬q_k ∧ ¬C) then                                   // FAIR
6:             return safe
7:         if CHECK-REACHABLE(I, T, ¬q_k ∧ ¬C) is safe then
8:             return safe
9:         s is the last state from GET-CEX-TRACE()                        // FAIR
10:        if CHECK-REACHABLE(T(s), T ∧ (W ↔ W'), s) is unsafe then
11:            return unsafe
12:        else
13:            D := GET-INVARIANT()
14:            W := W ∪ D
15:            g := GENERALIZING-NOLOOP(s, D)
16:            C := C ∪ g
17:        (I, T, ¬q_k) := IncreaseCounter(¬q_k)                           // k++
18:
19: function GENERALIZING-NOLOOP(s, D)
20:     assert( s ∈ ¬D and T(s) ⊆ D )
21:     assert(not IS-SAT(T ∧ ¬D', s))                                  // s ∧ T → D'
22:     g1 := GET-UC()
23:     assert(not IS-SAT(D, s))                                        // s → ¬D
24:     g2 := GET-UC()
25:     return g1 ∧ g2
```

- FAIR [9] tries to construct a lasso-shaped counterexample as follows: first, it searches for a candidate prefix (α); then, starting from the last (bad) state t of α, it searches for a suffix (β) that ends with t. Both steps are based on invariant checking. If the loop cannot be found, this bad state will be pruned. Fundamental optimizations include state generalization and, more importantly, extraction of *walls* (where, intuitively, states in a loop can only exist on one side of the wall). Then, FAIR iterates trying to find another candidate prefix for the lasso. The procedure terminates as soon as no prefix can be detected, in which case the property is proved.
- k-Liveness [13] tries to prove FGq based on the following intuition: if FGq holds, then there is a (finite) maximum number of times in which q can be violated in any path. The k-Liveness construction introduces a counter of the number of times q is violated and calls a safety checker to prove that the counter cannot exceed the given limit k. In case of failure, the limit is increased. k-Liveness proves the property if a k is found such that no path visits $\neg q$ more than k times. In general, k-Liveness is considered effective in *proving* the property. Notice, however, that k-Liveness – as described above – is incomplete, and will diverge if the property does not hold. On the other hand, it is possible to find counterexamples by checking the existence of

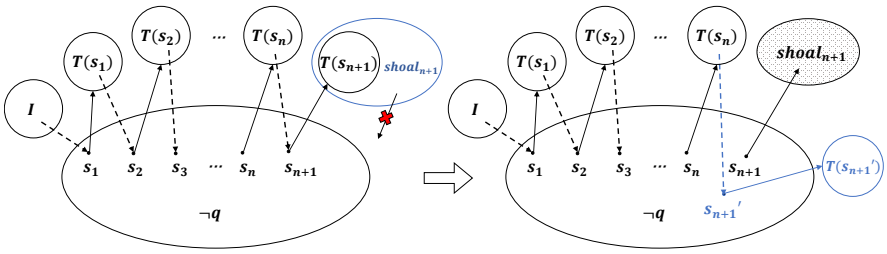

Fig. 1. Forward expansion and shoal construction (left); Rollback (right).

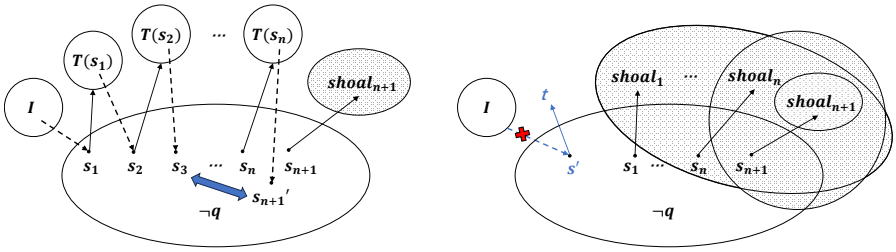

Fig. 2. Terminating conditions: counterexample found, unsafe (left); $\neg q$ no longer reachable from I, safe (right).

repeated bad states from the path returned by the safety-checking call. As already suggested in [13], k-Liveness can be run in parallel to bounded model checking [6], that is complete in the case of violation.

– k-FAIR [16] is a more recent approach, designed to inherit advantages from FAIR and k-Liveness. k-FAIR utilizes k-Liveness for proving correctness while leveraging FAIR for finding counterexamples. The k-FAIR algorithm is shown in Alg. 1. We see that FAIR and k-Liveness can both be considered a special case of k-FAIR. If line 17 is removed, the algorithm becomes FAIR. If line 5 and lines 9-16 are removed, it becomes k-Liveness.

3 Liveness checking with rlive

In this section we informally describe rlive, then present the pseudo code and some optimizations, and finally characterize its formal properties.

3.1 Overview

rlive is a new algorithm for liveness checking ($Sys \models FGq$). At a high level, rlive can be seen as a depth-first search with chronological backtracking and learning. rlive incrementally tries to build a counterexample to FGq, progressively extending it with more states in $\neg q$. In the *forward expansion* phase, rlive first

looks for a finite path π_1 from I to $\neg q$, with s_1 being the last state of π_1. Then, rlive looks for another path π_2 from $T(s_1)$ to $\neg q$, and so on. See Fig. 1, left. The forward expansion proceeds until one of two conditions holds.

1. if s_n is equal to s_i, with $i < n$, then a lasso-shaped counterexample exists, and the search terminates with *unsafe* (Fig. 2, left). The counterexample can be constructed by concatenating the previously found π_i.
2. if s_{n+1} cannot reach $\neg q$, then a *shoal* is built, i.e. a set of states closed under T and containing $T(s_{n+1})$, that can reach no target state ($shoal_{n+1}$ in Fig. 1, left). Clearly, no state in a shoal can belong to the counterexample; hence, shoals are learned and used to block the subsequent forward expansions.

In the second case, the algorithm *rolls back* to the previous level, and restarts the forward search, looking for a new way to enter $\neg q$. However, to avoid entering the shoals again, the target $\neg q$-state must have successors outside the shoals. (e.g. s'_{n+1} in Fig. 1, right). The algorithm terminates with *safe* whenever it rolls back to level 0, and finds no way to reach, from the initial states, the remaining subset of $\neg q$ while avoiding the shoal constraints (Fig. 2, right).

We remark that, upon backtracking, the forward search space is restricted to avoid the shoal constraints as well as the states in $\neg q$ that do not belong to the counterexample. Hence, the navigation toward the target is increasingly restricted because of the discovered shoal constraints and also because the target is progressively shrunk.

The algorithm described above is naturally implemented with primitives provided by the safety checker, such as deciding reachability and constructing the counterexamples and the invariants. A further practical optimization called *dead-state pruning*, trades off calls to the safety checker with calls to the SAT solver, enlarging the shoals with a cheap form of look ahead to further prune the target set.

3.2 Algorithm

Algorithm 2 describes how rlive is implemented using a generic invariant-checking engine implementing the API introduced in Section 2.3. To prove or falsify the liveness property FGq, rlive will maintain a global state set C at line 2, representing the shoals (i.e. states from which $\neg q$ can be reached only a finite number of times) discovered so far.

The algorithm starts from line 4, checking whether $\neg q$ is reachable from the initial states, using CHECK-REACHABLE. If it is not reachable, Gq is proved, and so FGq is verified. Otherwise, from the counterexample trace returned by CHECK-REACHABLE, we get a reachable $\neg q$-state s. Then the SEARCH-CEX function is called to search for the next $\neg q$-state from s.

When C is not empty, we block the states in C from the transition system by adding the constraint $\neg C \wedge \neg C'$ to T (lines 5 and 17). At the same time, the states to be searched become $\neg q \cap T^{-1}(\neg C)$, which ensures that the searched $\neg q$-states have $\neg C$ successors, to exclude the $\neg q$-states that are proved not to be part of a counterexample.

Algorithm 2 Implementation of rlive

```
1:  Liveness Property: FGq
2:  C := ∅      // shoals: ¬q can only be reached finitely-many times from states in C
3:
4:  procedure rlive(I, T, q)
5:      while CHECK-REACHABLE(I, T ∧ (¬C ∧ ¬C'), ¬q ∩ T^{-1}(¬C)) is unsafe do
6:          s is the reached ¬q-state from GET-CEX-TRACE()
7:          if SEARCH-CEX(s, ∅) then
8:              return unsafe
9:      return safe
10:
11: function SEARCH-CEX(s, B)
12:     if s ∈ B then                    // B: reachable ¬q-states from initial states
13:         return True
14:     while True do
15:         if PRUNE-DEAD(s) then
16:             return False
17:         if CHECK-REACHABLE(T(s), T ∧ (¬C ∧ ¬C'), ¬q ∩ T^{-1}(¬C)) is unsafe then
18:             t is the reached ¬q-state from GET-CEX-TRACE()
19:             if SEARCH-CEX(t, B ∪ {s}) then
20:                 return True
21:         else
22:             D := GET-INVARIANT()
23:             C := C ∪ D       // ¬q ∩ T^{-1}(¬C) cannot be reached from states in D
24:             return False
25:
26: function PRUNE-DEAD(s)
27:     while IS-SAT(s ∧ T ∧ ¬C') do
28:         μ := GET-MODEL()
29:         d := {l | l' ∈ μ}
30:         if not IS-SAT(T ∧ ¬C', d) then
31:             D := GET-UC( )        // unsatisfiable core returned by the SAT solver
32:             C := C ∪ D                  // states in D have no successor not in C
33:         else
34:             return False
35:     return True
```

In the SEARCH-CEX(s, B) function of line 11, the parameter s serves as a new reached $\neg q$-state, and the parameter B contains the $\neg q$-states that have been previously reached along the current trace. Therefore, in lines 12-13, when s has appeared in B, a lasso-shape counterexample has been found, so the function returns True (a counterexample has been detected). Line 15 is the implementation of an important heuristic called *dead-pruning*, which we describe in detail in the next subsection. A new call to CHECK-REACHABLE is performed to find the next $\neg q \cap T^{-1}(\neg C)$-state starting from the successor of s on line 17.The reason for searching from the $T(s)$-states is that s itself is a state that meets $\neg q \cap T^{-1}(\neg C)$. However, calculating the exact set $T(s)$ might be quite expensive,

so we use an overapproximation of $T(s)$, which we describe below in §3.3. If a state t can be reached, then the function is called recursively, t is used as the new starting state, and the s state is added to B. Otherwise, CHECK-REACHABLE would return an inductive invariant D on line 22. [4] This invariant is an overapproximation of the reachable states starting from $T(s)$, and none of these states can reach $\neg q \cap T^{-1}(\neg C)$. Therefore, states in D are shoals, so they can be added to C, and then the function returns False.

3.3 Optimizations

Avoiding the explicit computation of $T^{-1}(\neg C)$. When asking for the next $\neg q \cap T^{-1}(\neg C)$-state in the current trace, we can avoid the explicit computation of $T^{-1}(\neg C)$ by exploiting some additional knowledge about how the reachability engine CHECK-REACHABLE works. For example, if CHECK-REACHABLE is based on IC3 [8], we can simply add a constraint $T \wedge \neg C'$ to the SAT solver when asking for a $\neg q$-state.

Efficiently over-approximating $T(s)$. Using IC3 as an implementation of CHECK-REACHABLE allows us also to efficiently overapproximate the states $T(s)$ in the (recursive) searches for the next $\neg q$-states in the current trace (line 17). To do so, we slightly modify IC3,[5] and in particular the query that checks whether a given predecessor b of a bad ($\neg q$-)state intersects the initial states of the system. Rather than checking whether $T(s) \wedge b$ is satisfiable, we check the satisfiability of $s \wedge T$ under the assumption of b'. If the formula is unsat, we add the cube $c \subseteq b$ corresponding to the unsat core produced by the SAT solver (i.e. such that $c' =$ GET-UC()) to the 0-th frame of IC3. In this way, the 0-th frame of IC3 will effectively be our desired over-approximation of $T(s)$.

Dead states pruning. During rlive, lots of *dead states*, i.e. states that do not have any successors, are formed due to the strengthening of T and $\neg q$ using the discovered shoals. To prove that $\neg q$ cannot be reached from such a dead state, CHECK-REACHABLE needs to search for the predecessor states of $\neg q$ and describe the overapproximation of the reachable set from the dead state with the literals in the predecessors, which might require a large number of SAT queries.

Dead-pruning optimization is a simple and effective optimization (but probably not the only one) used to detect and quickly block the *dead states*. The optimization is used before calling CHECK-REACHABLE, to check whether a successor of the starting bad state is a dead state. If it is, then it can be excluded from the search and used to strengthen the shoals C.

[4] When the recursive call returns to the previous level, due to the incremental nature of the IC3, it can reuse the lemmas previously calculated in this level. However, we empirically found that such reuse doesn't result in an obvious boost in performance.

[5] Note that the same optimization can be applied also to other engines that use an "IC3-like" search, such as CAR.

Line 26 in Alg. 2 is the implementation of the *dead-pruning* heuristic. A successor d of s is computed on lines 27-29. If d is determined to be a dead state (line 30), then it can be added to C (after being generalized using the unsat core produced by the SAT solver). The function returns False once it finds a successor of s with successors outside of C. If all the successors of s are blocked as dead states, the function returns True.

3.4 Correctness Proof

This section presents the proofs for the correctness of rlive (Algorithm 2). We first show the following lemmas which are crucial for the proof.

Lemma 1. *Every state $t \in C$ can only reach a $\neg q$-state a finite number of times.*

Proof. According to Algorithm 2, C can be updated in either the SEARCH-CEX or PRUNE-DEAD procedure. Since the latter one is optional (it is an optimization), we first consider the proof without the PRUNE-DEAD procedure.

In the SEARCH-CEX procedure, C is the state set that is updated by the union of different inductive invariants returned by CHECK-REACHABLE (line 23), whose initial states are an over-approximation of successors of some $\neg q$-state s. From the correctness of CHECK-REACHABLE, every state t in the inductive invariant satisfies: (1) it may be reachable from the initial states (and $\neg q$-state s) due to over-approximation, thus may be reachable from s, and (2) it cannot reach the states in $\neg q \cap T^{-1}(\neg C)$ (line 17). By construction, assume $C = C_1 \cup C_2 \cup \ldots \cup C_n$ where C_k $(1 \leq k \leq n)$ is the k-th inductive invariant added into C. We prove the lemma by induction over n. Obviously, every state $t \in C_1$ cannot reach states in $\neg q$ (and $C_1 \cap \neg q = \emptyset$). So the lemma holds in the base case. For the inductive step (when $k > 1$), since every state $t \in C_k$ cannot reach $\neg q \cap T^{-1}(\neg(\bigcup_{1 \leq i \leq k-1} C_i))$, we consider a state $\tilde{s} \in \neg q$ in two different sets. If $\tilde{s} \in T^{-1}(\neg(\bigcup_{1 \leq i \leq k-1} C_i))$, t cannot reach $\tilde{s}$; otherwise, $\tilde{s} \notin T^{-1}(\neg(\bigcup_{1 \leq i \leq k-1} C_i))$ implies that $T(\tilde{s}) \subseteq (\bigcup_{1 \leq i \leq k-1} C_i)$, i.e., every successor of $\tilde{s}$ is in $(\bigcup_{1 \leq i \leq k-1} C_i)$. From the inductive hypothesis, every state in $(\bigcup_{1 \leq i \leq k-1} C_i)$ can only reach a $\neg q$-state a finite number of times. Therefore, we have that t can only reach a $\neg q$-state finitely-many times as well.

Taking the PRUNE-DEAD procedure into consideration, only those states whose successors are all in C are added into C (line 32). From the hypothesis assumption, every state in C can only visit a $\neg q$-state a finite number of times, so as the predecessors, those states can only visit a $\neg q$-state finitely-many times as well. □

Lemma 2. *Given $s \models \neg C$, when the* PRUNE-DEAD*(s) procedure returns, it returns True if and only if every successor of s, if existing, is in C.*

Proof. ($\Rightarrow$) The procedure returns True implies that either the SAT call at line 27 returns *unsat*, which indicates that every successor of s is in C, or there is some successor d of s that is not in C. However, since the procedure returns True, the SAT call at line 30 must return *unsat*, which indicates that every successor

of d, if existing, is in C. Then d will be added into C according to lines 31-32. So $d \in C$ becomes true. The above process will repeat inside the *while* loop at line 27 until every successor of s is in C.

($\Leftarrow$) If every successor of s is in C, the SAT call at line 27 will return *unsat*. Therefore, the *while* loop directly stops and the procedure returns True at line 35. □

Lemma 3. *1.* SEARCH-CEX(s, B) *returns True if and only if there is a lasso starting from* s *and its loop part contains a* $\neg q$*-state.*
2. SEARCH-CEX(s, B) *always terminates.*

Proof. 1. ($\Rightarrow$) The procedure is recursively implemented and it returns True as soon as a $\neg q$-state t (which can be the same as s) is already in B, indicating that a loop is detected. Moreover, t is reachable from the input state s, since t is detected from the successors of s by CHECK-REACHABLE. Therefore, a lasso starting from s and looping with t is found when the procedure returns True.
($\Leftarrow$) Assume the lasso is $s, \ldots, t_1, \ldots, (t_i, \ldots, t_j)$ in which $t_j = t_i$ $(1 \leq j \leq i)$ and every t_k $(1 \leq k \leq j)$ is a $\neg q$-state. First of all, we can prove that for each t_k, it is true that $t_k \in T^{-1}(\neg C)$, i.e., there is some successor of t_k that is not in C; otherwise, from t_k there cannot be a lasso looping with a $\neg q$-state, as based on Lemma 1, all successors of t_k being in C implies they can only visit a $\neg q$-state a finite number of times. Therefore, t_k can be found by the CHECK-REACHABLE call at line 17 and PRUNE-DEAD(t_k) cannot return True according to Lemma 2, implying that SEARCH-CEX(s, B) will not return False at line 16. As a result, SEARCH-CEX(s, B) will finally return True at line 13 once it finds t_j for the second time.
2. We prove that the *while* loop of line 17 of SEARCH-CEX(s, B) is terminating. The point is that the size of the state set $\neg q \cap T^{-1}(\neg C)$ keeps shrinking after each iteration of the loop, because the $\neg q$-state t at line 18 will be removed from $\neg q \cap T^{-1}(\neg C)$. The reason is that when the recursive SEARCH-CEX$(s, \emptyset)$ procedure returns False at line 19, the proof of Item 1 above guarantees that there is no lasso starting from t and looping with a $\neg q$-state. So C will be updated either by the inductive invariant (line 23) or the unsat core in the PRUNE-DEAD procedure (line 32) such that $t \notin T^{-1}(\neg C)$ is true, according to Lemmas 1 and 2. Therefore, t is successfully removed from $\neg q \cap T^{-1}(\neg C)$. In the worst case, the state set will become empty and CHECK-REACHABLE can terminate with *safe* as no bad state can be found at line 5.

□

Lemma 4. *1.* rlive(I, T, q) *always terminates.*
2. rlive(I, T, q) *returns safe if and only if the system* (I, T) *satisfies the property* FGq*.*

Proof. 1. The proof is analogous to that of Item 2 of Lemma 3, so it is omitted.
2. ($\Rightarrow$) Assume by contradiction that rlive returns safe, but the property doesn't hold. Therefore, there exists a lasso-shaped trace π of the form $s, \ldots, t_1,$

$\ldots, (t_i, \ldots, t_j)$ in which $t_j = t_i$ $(1 \leq j \leq i)$ and every $t_k (1 \leq k \leq j)$ is a $\neg q$-state. By Lemma 1, none of the states in π is in C, and moreover $t_i \in \neg q \cap T^{-1}(\neg C)$. Therefore, $s, \ldots, t_1$ is a trace reaching the bad state $\neg q \cap T^{-1}(\neg C)$ in the system $\langle X, I, T \wedge (\neg C \wedge \neg C') \rangle$, which is found by the CHECK-REACHABLE call at line 5. But then, SEARCH-CEX(t_1, $\varnothing$) at line 7 returns False by Lemma 3, and so rlive returns unsafe, which is a contradiction.

($\Leftarrow$) The system satisfies the property implies that every $\neg q$-state that is reachable from the initial states, if existing, can only be visited finitely-many times. Assume the number of such reachable $\neg q$-states is k ($k < +\infty$). If $k = 0$, the CHECK-REACHABLE procedure in the while loop of rlive (line 5) will directly return *safe* and thus rlive returns safe. When $k > 0$, assume the reachable $\neg q$-states are $s_1, \ldots, s_k$. So there are at most k iterations of the while loop, since each s_i $(1 \leq i \leq k)$ can be found at most once by the CHECK-REACHABLE call on line 5 (the argument is similar to the one used in the proof of Item 2 of Lemma 3). However, SEARCH-CEX($s_i, \emptyset$) will return False, because s_i can be visited only a finite number of times and thus no lasso can be detected. As a result, rlive cannot return unsafe inside the loop. And finally, rlive can only return safe in the worst case that every s_i is found and blocked in the while loop.

□

Theorem 1 (Correctness). rlive *can always terminate and terminate with the correct result.*

Proof. Directly from Lemma 4. □

4 Related Work

We have already introduced the main SAT-based liveness checking algorithms in Section 2.4. Here, we discuss their relation with rlive, highlighting both similarities and differences with our approach.

rlive *vs* L2S *[4].* The original liveness-to-safety transformation is conceptually very simple, and it can be applied with any off-the-shelf safety model checking algorithm, not necessarily based on SAT. The eager L2S transformation can however be inefficient, as it requires a duplication of the state variables, which might lead to significant performance penalties. In contrast, rlive follows a lazier approach, using an incremental reduction to safety, designed to exploit the invariant generation capability of modern SAT-based safety checking engines, which does not require duplicating the state variables and can be more efficient in practice.

rlive *vs* FAIR *[9].* At a high level, rlive and FAIR follow the same principle of incremental strengthening the input problem by exploiting the inductive invariants generated when refuting candidate counterexamples with a safety model checker. The main difference is in how the candidate counterexamples are identified and blocked: while FAIR does that by checking directly for looping paths

that start from a given reachable $\neg q$-state, rlive follows a more incremental approach, in which repeated (and recursive) safety checks are used to build a bad loop incrementally. As our experimental results show (see Section 5), this difference turns out to be crucial for performance in practice. A second difference regards the nature of the information extracted from the inductive invariants produced by the safety checker: in general, the walls of FAIR are regions that *cannot be crossed* to find a counterexample (i.e., all states of a counterexample to FGq are on one side of the wall), whereas shoals are regions that *must be avoided* completely (i.e., no state in a counterexample can part of a shoal).

rlive *vs* k-Liveness *[13].* The incremental approach used by rlive for constructing counterexamples is inspired by the k-Liveness algorithm; in some sense, rlive can in fact be seen as a depth-first (DFS) variant of k-Liveness, which performs instead a breadth-first (BFS) search (relative to the number k of times in which $\neg q$ can occur in the traces of the system). Thanks to its DFS approach, rlive doesn't need to maintain a global k value, but uses a different k for each trace; as such, it can sometims reach values of k which are beyond the capabilities of k-Liveness (see our results in §5).[6] Another difference between the two approaches is in the capability of finding counterexamples: although in principle complete, k-Liveness is more effective at proving properties than at disproving them, and already in the original paper [13] the authors recommend complementing it with BMC for finding counterexamples; on the other hand, rlive is effective both for safe and unsafe properties.

rlive *vs* k-FAIR *[16].* k-FAIR is a parametric combination of FAIR and k-Liveness, in which each candidate counterexample to FGq either is analyzed using FAIR, or causes an increase in the k counter of k-Liveness (see Algorithm 1). As such, the comparisons made above between rlive and FAIR or k-Liveness apply also to k-FAIR. Like k-FAIR, rlive can also be seen as trying to combine the strengths of the two techniques in a single algorithm; however, the two approaches differ significantly in how such integration is performed.

5 Evaluation

We have implemented rlive inside the nuXmv model checker [10]. Our implementation can use three different safety-checking engines, namely IC3, fCAR (Forward CAR), and bCAR (Backward CAR), relying on the latest version of CaDiCaL [7] as backend SAT solver. In this section, we experimentally evaluate rlive by comparing it with different state-of-the-art SAT-based liveness checking algorithms.

5.1 Experimental Setup

We include in our evaluation nuXmv [10] and IIMC [3], two state-of-the-art tools implementing SAT-based liveness-checking algorithms which are among the best-

[6] Note that here by k we mean the maximum recursion depth reached by rlive for a given candidate counterexample trace, as there is no explicit k counter in rlive.

Table 1. Tools and algorithms evaluated in the experiments.

Tools	Algorithms	Engines
nuXmv	rlive [-d]	IC3, fCAR, bCAR
	k-Liveness	
	FAIR	
	k-FAIR	
	L2S	
	k-Liveness + BMC	IC3, BMC
IIMC	k-Liveness	IC3
	FAIR	

performing ones in the most recent liveness-checking tracks of the Hardware Model Checking Competition (HWMCC) [1,2]. nuXmv implements L2S and k-Liveness, using a configuration that runs k-Liveness in lockstep with BMC as suggested in [13] for the latter (which we refer to as k-Liveness + BMC below). In addition to rlive, we also implemented other three liveness-checking algorithms on top of nuXmv, namely k-Liveness, FAIR, and k-FAIR. FAIR and k-FAIR are implemented according to Algorithm 1, and k-Liveness is added with the ability to find counterexamples by checking for repeated $\neg q$-states in the violated traces (before increasing the value of k). IIMC implements FAIR and "plain" k-Liveness instead (without BMC). Table 1 summarizes the tested tools, algorithms, and their engines. Regarding rlive, the '-d' flag is used to enable the dead-pruning optimization, otherwise rlive ignores lines 15-16 of Alg. 2.

We evaluate all the configurations on 223 benchmarks, in *aiger* [5] format, of the liveness property track of HWMCC 2015 and 2017 [1,2].[7] We ran the experiments on a cluster, which consists of Gold 6132 2.6GHz CPUs in 240 nodes running RedHat 4.8.5 with a total of 96GB RAM. For each test, we set the memory limit to 8GB and the time limit to 1 hour. During the experiments, each model-checking run has exclusive access to a dedicated node. For correctness checking, we compared the results from different solvers and found no discrepancies.

5.2 Experimental Results

Overview. The main results of the experiment are summarized in Table 2, in which the different tools/configurations are ordered by the total number of successfully solved instances within the given resource budget. From the table, we can see that rlive is the algorithm with the overall best performance in terms of the number of solved cases. More explicitly, rlive with the dead-pruning optimization and using IC3 as the backend solves the largest number of instances (159), and it is also the configuration that verifies the most cases (66). rlive is also the algorithm that finds the largest number of counterexamples, and this is

[7] Note that HWMCC editions after 2017 did not include a liveness track.

Table 2. Summary of overall results among different tools/configurations.

Configuration	#Solved	#Verified	#Violated
nuXmv -rlive -d	159	66	93
nuXmv -rlive -d (fCAR)	158	62	96
nuXmv -k-Liveness + BMC	146	61	85
nuXmv -rlive	145	54	91
nuXmv -rlive -d (bCAR)	142	45	97
nuXmv -L2S	139	65	74
nuXmv -k-Liveness	138	63	75
nuXmv -k-FAIR	124	54	70
IIMC -FAIR	82	47	35
nuXmv -FAIR	66	29	37
IIMC -k-Liveness	50	50	0

true for all configurations that we tested (with 'rlive -d' using bCAR being the best one).

Regarding other tools/algorithms, the best performing one is the k-Liveness + BMC implementation in nuXmv, solving a total number of 146 cases, which is 11% less than the best configuration of rlive (i.e., 'rlive -d'). All the other configurations solve significantly fewer instances than rlive.

The results in Table 2 also show that using different engines to run the rlive algorithm preserves good performance.[8] Under the same implementation platform, their overall performance is better than k-Liveness using IC3: 'rlive -d (fCAR)/ (bCAR)' solves 158/142 instances in total, while k-Liveness only solves 124. Using fCAR results in a much better performance than bCAR on verifying properties (62 vs. 45). However, applying the bCAR engine seems to be an advantage in finding counterexamples, although the gap with other engines is modest (and rlive in general performs very well on finding counterexamples).

Finally, the results show also the importance of the dead-pruning optimization. Before the dead-pruning optimization is enabled, the performance of rlive is similar to k-Liveness + BMC from nuXmv (145 vs. 146). Dead-pruning improves rlive (using IC3) by verifying 12 more instances and finding 2 additional violations.

Runtime efficiency. In order to evaluate the runtime efficiency of rlive, we show in Fig. 3 a plot on the number of solved instances (y-axis) in the given time limit (x-axis) for a subset of the tested configurations (all using IC3 as a backend). From the plot, it is evident that 'rlive -d' is significantly more efficient than the other competitors, always solving the largest number of instances within the timeout ranging from 600 seconds to 3600 seconds.

A more detailed comparison between rlive and other algorithms is shown in Fig. 4. From the plots, we can see that rlive outperforms other algorithms in a

[8] It should be mentioned that we also tried k-Liveness, FAIR and k-FAIR with the fCAR and bCAR engines in our tool as well, but they do not have better performance than using IC3. For page limit, we do not list the relevant data in the paper.

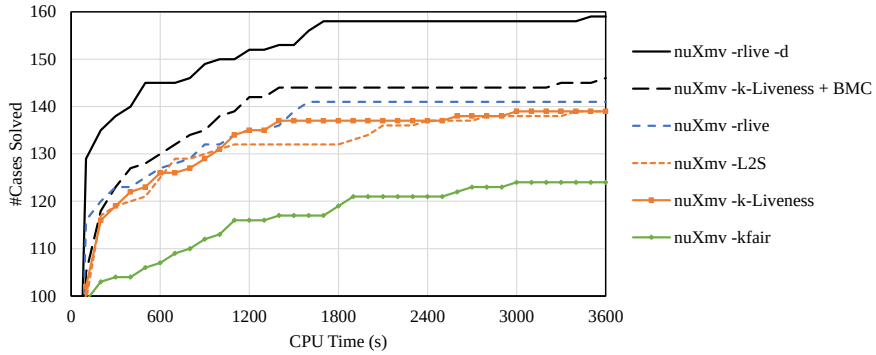

Fig. 3. Comparisons among the implementations under different configurations. (Note that for better readability the y-axis starts from a value of 100.)

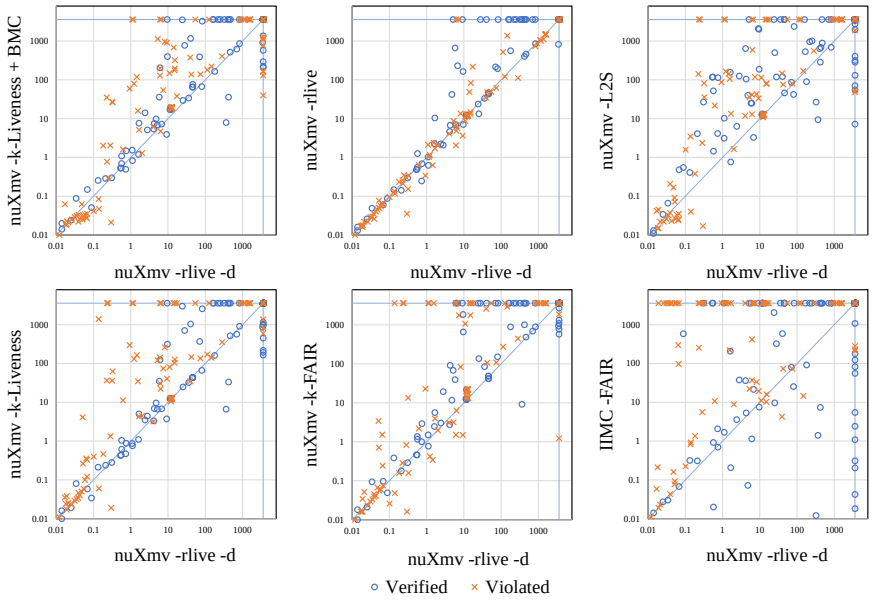

Fig. 4. Time comparison between rlive (with dead-pruning) and other implementations/configurations. rlive is always on the x-axis. Points above the diagonal indicate better performance of rlive. Points on the borders indicate timeouts (3600s).

large number of cases, especially in the case of violated properties. An interesting exception is IIMC-FAIR, which shows strengths that are complementary to those of rlive, particularly for verified properties.

Portfolio configurations. We analyze the behaviour of rlive in "portfolio" configurations, which is a technique often used in practice to improve performance

when multiple CPU cores are available. For this, we performed two (virtual) experiments. In the first experiment, we consider a (virtual) portfolio consisting of the algorithms using IC3 as the backend,[9] and compare it with (virtual) portfolios obtained by excluding a single algorithm at a time, in order to analyze the contribution of the excluded algorithm to the virtual best. The results are shown in Table 3. From the table, we can see that rlive contributes significantly to the performance of the virtual best, particularly for violated properties. Moreover, when multiple engines solve the same property, rlive is the fastest in the vast majority of cases (81 over 183 verified by the virtual best, with the 2nd best performing being the fastest only in 26 cases).

Table 3. Virtual Best results among implementations by IC3 engine. **VBS \ (Algorithm a)** refers to the removal of a from the portfolio, so the reduction in the number of solutions represents the contribution of a to the portfolio. **#Fastest Solution** represents the number of times algorithm a solves a case fastest in the full VBS portfolio.

Configuration	#Verified	#Violated	#Contribute	#Fastest Solution
VBS	82	101	-	-
VBS \ (nuXmv -rlive -d)	82	<u>85</u>	<u>17</u>	<u>81</u>
VBS \ (IIMC -FAIR)	<u>78</u>	101	4	20
VBS \ (nuXmv -k-Liveness + BMC)	82	97	4	14
VBS \ (nuXmv -L2S)	82	101	0	26
VBS \ (nuXmv -k-FAIR)	82	101	0	18
VBS \ (nuXmv -FAIR)	82	101	0	15
VBS \ (nuXmv -k-Liveness)	82	101	0	10

In the second experiment, we compose (virtual) portfolios in a "bottom up" way, by considering only configurations running two different algorithms in parallel. Also, in this case, the results in Table 4 clearly show the impact of rlive.

Table 4. Top 10 combinations of 2 algorithms implementation into one portfolio.

Configurations	#Solved	#Verified	#Violated
nuXmv -rlive -d & IIMC -FAIR	174	78	96
nuXmv -rlive -d & nuXmv -k-Liveness + BMC	172	71	101
nuXmv -rlive -d & nuXmv -L2S	172	76	96
nuXmv -rlive -d & nuXmv-k-Liveness	170	74	96
nuXmv -rlive -d & nuXmv-k-FAIR	166	71	95
nuXmv -rlive -d & nuXmv-FAIR	161	66	95
nuXmv -k-Liveness + BMC & nuXmv -L2S	161	76	85
nuXmv -k-Liveness + BMC & IIMC -FAIR	159	74	85
nuXmv -L2S & nuXmv -k-FAIR	152	76	76
nuXmv -L2S & nuXmv -k-Liveness	151	76	75

[9] We exclude 'nuXmv -rlive' because it is subsumed by 'nuXmv -rlive -d'.

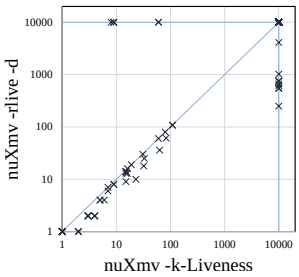

Fig. 5. Comparison of the k values (maximum recursion depths reached) at termination on property verified cases between k-Liveness and rlive. Points on the borders indicate timeouts.

Analysis of rlive behaviour. We explore the reasons for the excellent performance of rlive through Fig. 5, which compares the k value of k-Liveness to the corresponding maximum recursion depth in rlive on verified properties. They both represent that the algorithm can find a path containing at most k $\neg q$-states before terminating with safe. When both algorithms terminate within the time limit, the k value of rlive is always less (or equal) than the value of k-Liveness. Since k-Liveness performs a breadth-first search (in terms of k), it always needs to find the path that contains the most $\neg q$-states before it can terminate. On the other hand, the shoals generated by rlive during the search process help in blocking other $\neg q$-states, allowing rlive to converge at a smaller depth. In addition, rlive is better at solving cases where there is a path containing a large number of $\neg q$-states in the system, where k-Liveness needs to reach a very large k value to converge. These cases are located on the right border of Fig. 5. The recursion depths of rlive on these cases reach far over 100, with the deepest one reaching 4095. However, the maximum k value of k-Liveness is only around 100. Figure 5 shows also some cases (located in the upper border of the plot) which could be solved by k-Liveness but not by rlive. We investigated them and found that dead states caused the rollback steps of rlive to be slower. The current dead-pruning optimization, which only performs a one-step lookahead to discover dead states, is not effective for such instances (though in most cases this simple strategy works), suggesting future directions for improvement.

6 Conclusions

We presented rlive, a novel algorithm for the liveness checking problem FGq. The idea is to search for a lasso-shaped counterexample by repeatedly calling a safety checker to re-enter the $\neg q$ states set. The search proceeds in depth-first, backtracking when a state in $\neg q$ can be excluded by proving that $\neg q$ can only be reached finitely-many times from its successors, and cannot be part of a counterexample. The invariants returned by the underlying safety checker restrict the search progressively. We called such invariants shoals, as intuitively they represent states that must be avoided when searching for a counterexample. A thorough experimental evaluation clearly demonstrates that rlive is superior to the other liveness checkers, both in terms of benchmarks solved and run time.

Regarding future research, we plan to extend this work in several directions. First, we will investigate heuristics to control the exploration order of bad states and the counterexamples produced by the safety checker. Second, we will con-

sider the extraction of proofs from rlive. Third, we will consider extensions of rlive to the infinite-state case, in combination with algorithms for finding non-lasso-shaped counterexamples such as [11].

Acknowledgment. We thank anonymous reviewers for their helpful comments. Yechuan Xia and Jianwen Li are supported by the National Natural Science Foundation of China (Grant #62372178 and #U21B2015), "Digital Silk Road" Shanghai International Joint Lab of Trustworthy Intelligent Software under Grant 22510750100, and Shanghai Collaborative Innovation Center of Trusted Industry Internet Software. A. Cimatti and A. Griggio acknowledge the support of the PNRR project FAIR - Future AI Research (PE00000013), under the NRRP MUR program funded by the NextGenerationEU, and of the PNRR MUR project VITALITY (ECS00000041), Spoke 2 ASTRA - Advanced Space Technologies and Research Alliance.

References

1. HWMCC 2015. `http://fmv.jku.at/hwmcc15/`
2. HWMCC 2017. `http://fmv.jku.at/hwmcc17/`
3. IIMC. `https://github.com/mgudemann/iimc`
4. Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. In: Proc. 7th Int. Workshop on Formal Methods for Industrial Critical Systems. Electronic Notes in Theoretical Computuer Science, vol. 66:2 (2002)
5. Biere, A.: AIGER Format. `http://fmv.jku.at/aiger/FORMAT`
6. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 193–207. Springer Berlin Heidelberg, Berlin, Heidelberg (1999)
7. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In: Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions. Department of Computer Science Report Series B, vol. B-2020-1, pp. 51–53. University of Helsinki (2020)
8. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) Verification, Model Checking, and Abstract Interpretation, pp. 70–87. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
9. Bradley, A.R., Somenzi, F., Hassan, Z., Zhang, Y.: An incremental approach to model checking progress properties. In: FMCAD. pp. 144–153. FMCAD Inc. (2011)
10. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: CAV. Lecture Notes in Computer Science, vol. 8559, pp. 334–342. Springer (2014)
11. Cimatti, A., Griggio, A., Magnago, E.: LTL falsification in infinite-state systems. Inf. Comput. **289**(Part), 104977 (2022)
12. Claessen, K., Eén, N., Sterin, B.: A circuit approach to LTL model checking. In: FMCAD. pp. 53–60. IEEE (2013)
13. Claessen, K., Sörensson, N.: A liveness checking algorithm that counts. In: FMCAD. pp. 52–59. IEEE (2012)
14. Clarke, E.M., Grumberg, O., Hamaguchi, K.: Another look at LTL model checking. Formal Methods in System Design **10**(1), 47–71 (1997)

15. Een, N., Mishchenko, A., Brayton, R.: Efficient implementation of property directed reachability. In: Proceedings of the International Conference on Formal Methods in Computer-Aided Design. pp. 125–134. FMCAD '11, FMCAD Inc, Austin, Texas (2011)
16. Ivrii, A., Nevo, Z., Baumgartner, J.: k-fair = k-liveness + FAIR revisiting sat-based liveness algorithms. In: FMCAD. pp. 1–5. IEEE (2018)
17. Jovanovic, D., Dutertre, B.: Property-directed k-induction. In: Formal Methods in Computer-Aided Design. pp. 86–92 (2016)
18. Li, J., Zhu, S., Zhang, Y., Pu, G., Vardi, M.Y.: Safety model checking with complementary approximations. In: Proceedings of the 36th International Conference on Computer-Aided Design. pp. 95–100. ICCAD '17, IEEE Press (2017)
19. Marques-Silva, J., Lynce, I., Malik, S.: Conflict-driven clause learning sat solvers. Handbook of satisfiability **185** (2009)
20. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt, W.A., Somenzi, F. (eds.) Computer Aided Verification, pp. 1–13. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
21. McMillan, K.L.: Applying SAT methods in unbounded symbolic model checking. In: Proceedings of the 14th International Conference on Computer Aided Verification. pp. 250–264. CAV '02, Springer-Verlag, Berlin, Heidelberg (2002)
22. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: DAC. pp. 530–535. ACM (2001)
23. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (sfcs 1977). pp. 46–57 (Oct 1977)
24. Sheeran, M., Singh, S., Stalmarck, G.: Check safety properties using induction and a SAT-solver. In: Proc. 3rd Int. Conf. on Formal Methods in Computer-Aided Design. Lecture Notes in Computer Science, vol. 1954, pp. 108–125. Springer (2000)
25. Silva, J.P.M., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. IEEE Trans. Computers **48**(5), 506–521 (1999)
26. Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. In: Banff Higher Order Workshop. LNCS, vol. 1043, pp. 238–266. Springer (1995)
27. Vizel, Y., Gurfinkel, A.: Interpolating property directed reachability. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification. pp. 260–276. Springer International Publishing, Cham (2014)