Infinite-state Liveness Checking with rlive

Alessandro Cimatti[1], Alberto Griggio[1], Christopher Johannsen[2], Kristin Yvonne Rozier[2], and Stefano Tonetta[1]

[1] Fondazione Bruno Kessler, Trento, Italy
[2] Iowa State University, USA

Abstract. rlive is a recently-proposed SAT-based liveness model checking algorithm that showed remarkable performance compared to other state-of-the-art approaches, both in absolute terms (solving more problems overall than other engines on standard benchmark sets) as well as in relative terms (solving several problems that none of the other engines could solve). rlive proves or disproves properties of the form FGq, by trying to show that $\neg q$ can be visited only a finite number of times via an incremental reduction to a sequence of reachability queries. A key factor in the good performance of rlive is the extraction of "shoals" from the inductive invariants of the reachability queries to block states that can reach $\neg q$ a bounded number of times.
In this paper, we generalize rlive to handle infinite-state systems, using the Verification Modulo Theories paradigm. In contrast to the finite-state case, liveness cannot be simply reduced to finding a bound on the number of occurrences of $\neg q$ on paths. We propose therefore a solution leveraging predicate abstraction and termination techniques based on well-founded relations. In particular, we show how we can extract shoals that take into account the well-founded relations. We implemented the technique on top of the open source VMT engine IC3ia and we experimentally demonstrate how the new extension maintains the performance advantages (both absolute and relative) of the original rlive, thus significantly contributing to advancing the state of the art of infinite-state liveness verification.

1 Introduction

Liveness checking is the problem of proving (or disproving) that a property of the form $\mathbf{FG}q$ holds in a given transition system S. When $S \models \mathbf{FG}q$, all the traces of S are such that q eventually stabilizes, i.e., it holds indefinitely from a certain state on (equivalently, $\neg q$ is visited only a finite number of times). Dually, when $S \not\models \mathbf{FG}q$, there exists an infinite path π satisfying $\mathbf{GF}\neg q$ that hits condition $\neg q$ (usually called a fairness condition) an infinite number of times. Liveness checking is a fundamental enabler for verification, since LTL model checking can be reduced to liveness checking thanks to standard techniques [17].

In this paper, we tackle the problem of liveness checking for infinite-state systems. In contrast to the finite-state case, the problem cannot be simply reduced to the search for lasso-shaped fair paths, as an infinite-state transition system

may only admit violations that cannot be presented as lasso-shaped paths. Dually, even if the property holds, there may be no upper bound on the number of times the fairness condition is satisfied along any path.

Our starting point is rlive, a recently-proposed SAT-based liveness checking algorithm for the finite state case, that demonstrated remarkable performance compared to other state-of-the-art approaches [47]. The rlive algorithm proves or disproves properties of the form $\mathbf{FG}q$, by trying to show that $\neg q$ can be visited only a finite number of times via an incremental reduction to a sequence of reachability queries. A distinguishing feature of rlive is the extraction of "shoals" from the inductive invariants of the reachability queries, to block states that can reach $\neg q$ only a bounded number of times.

We generalize rlive to handle infinite-state systems, using the Verification Modulo Theories (VMT) paradigm [11]. In contrast to the finite-state case, liveness cannot be reduced to finding a bound on the number of occurrences of $\neg q$ that is global for all the paths. We propose therefore a solution integrating predicate abstraction and termination techniques based on well-founded relations.

At the top level, our new algorithm, which we call rlive-inf, can be seen as a counterexample-guided abstraction refinement loop (CEGAR) [29] that maintains a set of predicates inducing an abstract state space and a set of well-founded relations. The procedure enumerates *abstract lassos*, which are candidate counterexample traces containing a repeated abstract state satisfying the fairness condition $\neg q$. Well-founded relations are used to avoid discovering candidate loops that can be proved to be terminating. If an abstract lasso is returned, rlive-inf attempts to concretize it and, in case of failure, it refines the abstraction by finding more predicates and/or well-founded relations. A key non-trivial step in the generalization of rlive to the infinite-state case is the construction of shoals that take into account the well-founded relations to block states that can reach $\neg q$ a finite but potentially unbounded number of times.

We implemented rlive-inf on top of the open-source VMT engine ic3ia [34], hence obtaining a fully-symbolic LTL model checker for infinite-state transition systems. We evaluated our implementation on a wide set of benchmarks from the literature, and compared it with αL2S [23], the state-of-the-art technique based on abstract liveness-to-safety, which is also implemented in ic3ia. Our experimental evaluation clearly demonstrates the value of rlive-inf. First, rlive-inf solves more benchmarks than αL2S, both safe and unsafe. Second, rlive-inf is on average significantly faster than αL2S. Interestingly, the two approaches are quite complementary, in the sense that the virtual best solver is significantly more effective than both procedures alone.

Structure of the paper. In Section 2 we present some background, and in Section 3 we discuss the liveness checking problem. We present the rlive-inf algorithm in Section 4, and we prove its correctness in Section 5. In Section 6 we discuss its limitations, and in Section 7 we compare it to other approaches for liveness checking. In Section 8 we present our experimental evaluation, and in Section 9 we draw conclusions and present some directions for future work.

2 Preliminaries

We focus on model checking of infinite state systems described with symbolic formulas. We work in the setting of SMT [2] to interpret formulas modulo a given background theory and of VMT [11] to describe states and transitions with SMT formulas.

A symbolic transition system is defined as $S := \langle X, I, T\rangle$ where X is a set of variables, $I(X)$ is a one-state formula defining the initial condition and $T(X, X')$ is a two-state formula defining the transition relation, where $X' = \{x' \mid x \in X\}$ is the set of next-state variables. For a formula ϕ, we denote ϕ' the formula obtained by replacing each $x \in X$ with its next-state variant $x' \in X'$ in ϕ. A state is an assignment to the variables of X. A finite path π of S is a finite sequence of states $s_0, s_1 \ldots, s_n$ such that $s_0 \models I$ and $s_i, s'_{i+1} \models T$ (and similarly for an infinite path $s_0, s_1, \ldots$). If π is a finite path of the form $s_0, \ldots, s_n$, we refer to s_0 as π_{first} and to s_n as π_{last}. A state is reachable in S if it occurs in a finite path of S. An infinite path is lasso-shaped if it can be expressed as $\alpha \cdot \beta^\omega$, where the stem $\alpha := s_0, s_2, \ldots, s_l$ and the loop $\beta := s_{l+1}, \ldots, s_k$ are finite sequences of states in S such that $s_0, \ldots, s_k$ is a finite path of S and $s_k = s_{l+1}$.

Invariant Checking The invariant checking problem asks whether a state property $P(X)$ holds in all reachable states of a system $S = \langle X, I, T\rangle$. An invariant checking procedure $\mathsf{check\text{-}inv}(X, I, T, P)$ returns a finite trace $\pi := s_0, s_1, \ldots, s_n$ of S where $s_i \models \neg P$ for some $0 \leq i \leq n$, or an inductive invariant inv such that $inv \models P$, $I \models inv$, and $inv \wedge T \models inv'$. See [13] for details. Many techniques exist to perform invariant checking of infinite-state systems, e.g. [13,37,33,4,43,36,5,38]. In the following, we assume a symbolic transition system $S = \langle X, I(X), T(X, X')\rangle$ is given.

Predicate Abstraction Predicate abstraction is defined over a set of predicates $\mathbb{P} = \{\gamma_1(X), \ldots, \gamma_n(X)\}$ [28,15]. For each predicate $\gamma_i(X)$ we assume a corresponding a Boolean variable $\widehat{\gamma_i}$. With an abuse of notation, we may also denote $\{\widehat{\gamma_1}, \ldots, \widehat{\gamma_n}\}$ with $\mathbb{P}$. We relate each predicate to its abstracted Boolean variable with the following formula:

$$\textsc{PrDef}(X, \mathbb{P}) = \bigwedge_i \widehat{\gamma_i} \leftrightarrow \gamma_i(X)$$

The predicate abstraction of S, denoted $\widehat{S} := \langle \widehat{X}, \widehat{I}, \widehat{T}\rangle$, is defined as:

$$\widehat{X} := \mathbb{P} \qquad \widehat{I} := \exists X.(I(X) \wedge \textsc{PrDef}(X, \mathbb{P}))$$
$$\widehat{T} := \exists X.(T(X, X') \wedge \textsc{PrDef}(X, \mathbb{P}) \wedge \textsc{PrDef}(X', \mathbb{P}'))$$

Given a state s, its abstraction $\widehat{s}$ with respect to $\mathbb{P}$, referred to as $\mathbb{P}$-abstraction, is defined as $\{\widehat{\gamma_i} \mid s \models \gamma_i\} \cup \{\neg\widehat{\gamma_j} \mid s \not\models \gamma_j\}$. Given an abstract state $\widehat{s}$, its set of corresponding concrete states is denoted as $[\![\widehat{s}]\!]$: these are all the states that satisfy the formula $\bigwedge_{\widehat{\gamma_i} \in \widehat{s}} \gamma_i \wedge \bigwedge_{\neg\widehat{\gamma_j} \in \widehat{s}} \neg\gamma_j$.

Disjunctive Well-founded Relations A *well-founded relation* $\rho \subseteq Q \times Q$ is a binary relation such that every non-empty subset $U \subseteq Q$ has a minimal element with respect to ρ, that is, there is some $m \in U$ such that no $u \in U$ satisfies $\rho(u, m)$. A *disjunctive* well-founded relation is defined as a finite union of well-founded relations.

Termination of a program can be proven by finding a well-founded relation for the program's states. In order to reason about general transition relations (including those with disjunctions) we use disjunctive well-founded relations. One way to obtain a well-founded relation for a program with non-disjunctive transition relation T is via a ranking function $r(X)$ that assigns a natural number to each program state such that the relation $\{(r(s_0), r(s_1)) \mid s_0, s_1' \models T\}$ is well-founded. Various techniques exist to synthesize ranking functions, e.g. [40,30,22,45].

3 Liveness Checking

The problem of model checking general LTL properties [17] can be reduced, following standard techniques (e.g., [46,17,7]), to the problem of model checking a property of the form $\mathbf{FG}q$, where q is a state formula over X. The problem of checking whether $\mathbf{FG}q$ holds in S, denoted $S \models \mathbf{FG}q$, amounts to checking if q eventually stabilizes on every path of S, i.e., for all $\pi \in S. \exists i. \forall j \geq i. \pi[j] \models q$. The dual problem is checking the existence of an infinite path $\pi \in S$ satisfying $\mathbf{GF}\neg q$, that is, visits $\neg q$ an infinite number of times. $\neg q$ may be referred to as the fairness condition, and π as a fair path. In the following, we assume the $\mathbf{FG}q$ property as given. Several algorithms for liveness checking have been proposed in the past, including Liveness-to-safety (L2S) [3], FAIR [8], k-liveness [16], k-FAIR [35], rlive [47]. We provide details on the algorithms most relevant to our contributions here.

rlive The rlive algorithm [47] performs a depth-first search for a loop violating $\mathbf{FG}q$, i.e., a fair path π satisfying $\mathbf{GF}\neg q$, hitting $\neg q$ an infinite number of times. rlive (see Algorithm 1) incrementally performs a series of invariant model checking queries. If the algorithm determines that $\neg q$ is unreachable from the current state in the search, rlive adds the newly-found inductive invariant to a set of states known as *shoals*, otherwise it reaches another state s that satisfies $\neg q$. A *shoal* is a set of states that can only reach $\neg q$ a finite number of times. If s is already on the search stack B, then rlive has found a loop satisfying $\mathbf{GF}\neg q$, so the algorithm terminates with *Unsafe*. Otherwise rlive adds this state to the stack B and continues the search. The algorithm terminates with *Safe* once it shows that the initial set of states is contained in the shoals.

Liveness to Safety When S is finite, $S \not\models \mathbf{FG}q$ if and only if there exists a *lasso-shaped* path $\alpha \cdot \beta^\omega$ in S where some state $b \in \beta$ is such that $b \models \neg q$. The L2S transformation [3] is an approach for reducing the problem of checking for

Algorithm 1: rlive algorithm for finite-state systems for the property $\mathbf{FG}q$ with variable set X, initial condition I, and transition relation T.

```
1  Procedure rlive(X, I, T, FGq) begin
2      C := ⊥
3      B := empty stack of states
4      while check-inv(X, I, T ∧ (¬C ∧ ¬C′), T^{-1}(¬C) → q) is Unsafe do
5          s := final state of get-mc-cex()
6          B.push(s)
7          while B is not empty do
8              s := B.top()
9              if check-inv(X, T(s), T ∧ (¬C ∧ ¬C′), T^{-1}(¬C) → q) is Unsafe
                then
10                 t := final state of get-mc-cex()
11                 if t ∈ B then
12                     return Unsafe
13                 B.push(t)
14             else
15                 inv := get-mc-inv()
16                 C := C ∨ inv
17                 B.pop()
18     return Safe
```

the existence of a lasso-shaped path violating q to that of checking an invariant property. The idea is to record the first state of a loop satisfying $\mathbf{GF}\neg q$ by introducing a copy X_c of the state variables X of S and a fresh variable SVD (saved) to record that the loop has started. X_c is nondeterministically assigned a state violating q (the start of the loop) and never changed after that, and the search tries to reach a state where each state variable has the same value as its copy and SVD is true, which implies that a violating lasso is detected.

Abstract Liveness to Safety If S is infinite, the existence of a loop $\alpha \cdot \beta^\omega$ proves that $S \not\models \mathbf{FG}q$, but there may be other non lasso-shaped counterexamples. This makes the L2S construction incomplete, in the sense that it is no longer guaranteed to find a counterexample to $S \models \mathbf{FG}q$ if one exists, and unsound, in that $S \models \mathbf{FG}q$ does not follow from proving the absence of lasso-shaped violations. One possibility for restoring soundness is to prove the absence of lasso-shaped counterexamples in a finite abstraction of the input system, e.g., one induced by a finite set of predicates $\mathbb{P}$.

Given $\mathbb{P}$, the abstract liveness-to-safety (αL2S) encoding [23] consists of storing only the truth assignments to the predicates non-deterministically, and detecting a loop if the system visits again the same abstract state violating q. Predicate abstraction is however not complete in general for proving liveness properties of infinite-state systems in the sense that, even if the property is satisfied by the concrete system, there may be no finite set of predicates such that the abstraction does not contain counterexamples. In [23], predicate abstraction

Algorithm 2: rlive-inf algorithm for infinite-state liveness checking of the property $\mathbf{FG}q$ with variable set X, initial condition I, transition relation T.

```
1 Procedure rlive-inf(X, I, T, FGq) begin
2     ℙ := predicate symbols in I, q
3     𝕎 := ∅ // set of well-founded relations
4     while abs-rlive-wfr(X, I, T, q, ℙ, 𝕎) is Unsafe do
5         π̂ := get-abs-cex()
6         if feasible(I, T, ℙ, π̂) then
7             return Unsafe
8         𝕎, ℙ := refine(I, T, ℙ, π̂)
9     return Safe
```

was combined with arguments based on well-founded relations to strengthen the proof power of the abstraction, and integrated in a CEGAR loop [18] to improve the precision of the abstraction (by discovering either new predicates or new well-founded relations) upon discovery of spurious counterexamples. The resulting algorithm performs very well in practice, outperforming alternative approaches on several benchmarks.

4 Infinite state rlive

In this section we present our generalization of rlive to the infinite-state case, obtained by leveraging predicate abstraction and well-founded relations. We use predicate abstraction to address the infiniteness of the state space and well-founded relations to address abstract paths of the form $\widehat{\alpha} \cdot \widehat{\beta}^{\omega}$ that are not feasible in S, even though every finite unrolling $\widehat{\alpha} \cdot \widehat{\beta} \ldots \widehat{\beta}$ is feasible.

4.1 High-level CEGAR loop

At the high level, rlive-inf implements a CEGAR loop that at each iteration generates and solves an *abstract liveness checking* problem using abs-rlive-wfr, refining the precision of the abstraction upon detection of spurious counterexamples. The pseudocode for the main rlive-inf procedure is shown in Algorithm 2. The algorithm consists of the following main ingredients:

Abstraction Precision The *precision* for the abstraction consists of a finite set of predicates $\mathbb{P}$ and a finite set of well-founded relations $\mathbb{W}$. Initially, $\mathbb{P}$ contains all predicate symbols in I and q, and $\mathbb{W}$ is empty.

Problem Definition At each iteration of the main loop of line 4, the procedure abs-rlive-wfr is called to solve the following problem.

Definition 1 (Abstract Liveness Checking Problem). *Given a variable set X, initial condition I, transition relation T, system $S := \langle X, I, T\rangle$, property $P := \boldsymbol{FG}q$, predicates $\mathbb{P}$, and set of well-founded relations $\mathbb{W}$, we call* abstract liveness checking *the problem of checking whether there exists a* finite sequence $\pi^0, \ldots, \pi^l$ *of* finite paths *(called* segments*) of S such that:*

- ${\pi^0}_{\mathsf{first}} \models I$, ${\pi^l}_{\mathsf{last}} \models \neg q$, *and the segments can be concatenated to form an* abstract path π, *that is, for all* $0 \leq m < l$, $\widehat{\pi^m{}_{\mathsf{last}}} = \widehat{\pi^{m+1}{}_{\mathsf{first}}}$;
- *the path π contains four states s_i, $s_j \equiv {\pi^l}_{\mathsf{last}}$, s_h, and s_k (with $i \leq h < k \leq j$) such that:*
  1. $s_i \models \neg q$, $s_j \models \neg q$, $s_h \models \neg q$, $s_k \models \neg q$;
  2. $\widehat{s_i} = \widehat{s_j}$; *and*
  3. *for all* $W \in \mathbb{W}$, $(s_h, s_k) \notin W$.

If such paths $\pi^0, \ldots, \pi^l$ exist, we say that the abstract liveness checking problem is unsafe, and call the path π an abstract lasso.

If abs-rlive-wfr cannot find an abstract lasso, then the property holds, and rlive-inf returns *Safe*. [3] Otherwise, the feasible procedure of line 6 checks whether the abstract lasso corresponds to at least one concrete lasso-shaped path in S in which $\neg q$ holds at least once in the loop; if this is the case, rlive-inf finds a counterexample to $\mathbf{FG}q$, and *Unsafe* is returned.

Refinement If the abstract lasso is infeasible, the refine procedure is invoked at line 8 to try to improve the precision of the abstraction, by discovering new predicates and/or new well-founded relations. The method is the same as that used in [23]. Specifically, the abstract lasso can be spurious for two reasons:

1. either there exists a finite unrolling of the abstract lasso that is infeasible in the concrete system S; or
2. the looping part of the abstract lasso cannot be executed infinitely often even though all its finite unrollings are feasible.

In the first case, refine generates new predicates to add to $\mathbb{P}$ using Craig interpolation [32], whereas in the second case, refine tries to generate both new predicates and new well-founded relations using ranking function synthesis techniques from termination analysis [30]. Since the approach in [30] only admits transition relations without disjunctions that represent "simple lassos" of the form $\varphi_{stem} \wedge \varphi_{loop}$, we enumerate all simple lassos symbolically represented by the candidate counterexample using [39] and synthesize a ranking function for each. See Section 4.2 of [23] for details. Note that, in general, refinement might fail (because both Craig interpolation and ranking function synthesis techniques are incomplete for some theories); in such cases, rlive-inf will diverge.

[3] Essentially, Definition 1 ensures that the absence of abstract lassos implies the disjunctive well-foundedness of the transitive closure of the transition relation of S [19].

Algorithm 3: abs-rlive-wfr procedure for fairness conditions q with variable set X, initial condition I, transition relation T, predicate set $\mathbb{P}$, and well-founded relation set $\mathbb{W}$.

```
1  C := ⊥ // global persistent cache of blocked states (shoals)
2  Procedure abs-rlive-wfr (X, I, T, q, ℙ, 𝕎) begin
3      B̂ := empty stack of abstract states
4      while check-inv(X, I, T ∧ (¬C ∧ ¬C′), T⁻¹(¬C) → q) is Unsafe do
5          ŝ := ℙ-abstraction of final state of get-mc-cex()
6          B̂.push(ŝ)
7          while B̂ is not empty do
8              ŝ := B̂.top()
9              X_c := {x_c | x ∈ X}, SVD := fresh-bool-var()
10             X̄ := X ∪ X_c ∪ {SVD}
11             T̄ := T ∧ ((¬SVD ∧ SVD′) → ((X = X_c) ∧ ¬q)) ∧ (X′_c = X_c) ∧ (SVD →
                   SVD′)
12             P̄ := (T̄⁻¹(¬C) ∧ SVD ∧ ¬𝕎(X_c, X)) → q
13             if check-inv(X̄, [[ŝ]] ∧ ¬SVD, T̄ ∧ (¬C ∧ ¬C′), P̄) is Unsafe then
14                 t̂ := ℙ-abstraction final state of get-mc-cex()
15                 if t̂ ∈ B̂ then
16                     return Unsafe
17                 B̂.push(t̂)
18             else
19                 B̂.pop()
20                 inv̄ := get-mc-inv() // includes predicates over X_c, SVD
21                 C_new := (∃X_c, SVD. (inv̄ ∧ ¬SVD))
22                 C := C ∨ C_new
23     return Safe
```

4.2 Abstract liveness checking

We now describe the core of rlive-inf: abstract liveness checking with the abs-rlive-wfr procedure. Algorithm 3 provides the pseudocode of the abs-rlive-wfr procedure, solving the abstract liveness checking problem in Definition 1. Conceptually, the procedure consists of two main blocks:

Segmented Search abs-rlive-wfr translates the abstract liveness checking problem of Definition 1 into a *sequence* of invariant checking problems, in which candidate abstract lassos are constructed incrementally (segment-by-segment), with a depth-first search for successor $\neg q$-states from the current $\neg q$-state, analogously to how the original rlive works in the finite-state case.

Caching abs-rlive-wfr maintains a (symbolic) *cache* of states that cannot be part of any counterexample, i.e., that cannot be part of any path of S satisfying $\neg q$ infinitely often. Such cache (made of states belonging to shoals [47]), is global

and persistent across different calls of abs-rlive-wfr at different iterations of the main CEGAR loop of Algorithm 2, ensuring that once a set of states is blocked by rlive-inf, it is never considered again in the future when searching for abstract lassos.

As compared to finite-state rlive (Algorithm 1), the high-level structure is largely the same with the following key differences:

Abstract Search Stack The search stack $\widehat{B}$ is a stack of $\mathbb{P}$-abstract states.

Well-founded Relation Reasoning When searching for an abstract lasso, the algorithm checks for path segments that include a pair of $\neg q$-states not in any relation in $\mathbb{W}$. It does so with an extended system encoding (lines 10, 11) and modified property (line 12), explained in detail below.

Shoal Construction When constructing shoals, the extra variables introduced by the extended system encoding are removed from the returned inductive invariant, by first forcing SVD to false and then by existentially eliminating them (line 21).

The algorithm returns *Unsafe* if it finds an abstract $\neg q$-state $\widehat{t}$ already in the search stack $\widehat{B}$, implying the existence of an abstract lasso. Crucially, the check of $\overline{P}$ on line 13 also implies the existence of at least one pair of $\neg q$-states within that abstract lasso not in any relation in $\mathbb{W}$. The resulting counterexample will then be checked for feasibility either due to the imprecision of the $\mathbb{P}$-abstraction or the existence of a well-founded relation proving the counterexample's finiteness by the outer CEGAR procedure.

Otherwise, if no $\neg q$-state pair not in any relation in $\mathbb{W}$ is reachable from the current top element of $\widehat{B}$, the inductive invariant from the corresponding model-checking call is added to the shoals C (lines 21-22). The algorithm returns *Safe* if there are no more $\neg q$-states outside the shoals that are reachable from the initial states.

Extended System Encoding The encoding used to find path segments with pairs of states not in any well-founded relation in $\mathbb{W}$ is inspired by the L2S encoding. Specifically, it introduces a new variable SVD and a copy of the state variables X_c, with SVD initially set to false. The transition relation then enforces:

1. SVD is non-deterministically set to true in a state where $\neg q$ holds and X_c takes on the value of X: $((\text{SVD}' \wedge \neg\text{SVD}) \rightarrow ((X = X_c) \wedge \neg q))$.
2. Once SVD is true, it stays true: $(\text{SVD} \rightarrow \text{SVD}')$.
3. The variables in X_c are frozen: $(X_c' = X_c)$.

Putting this all together, the *extended system* $\overline{S} := \langle \overline{X}, \overline{I}, \overline{T} \rangle$ is defined as:

$$\begin{aligned}
\overline{X} &:= X \cup \{\text{SVD}\} \cup X_c \\
\overline{I} &:= I \wedge \neg\text{SVD} \\
\overline{T} &:= T \wedge ((\text{SVD}' \wedge \neg\text{SVD}) \rightarrow ((X = X_c) \wedge \neg q)) \wedge \\
&\qquad (X_c' = X_c) \wedge (\text{SVD} \rightarrow \text{SVD}')
\end{aligned}$$

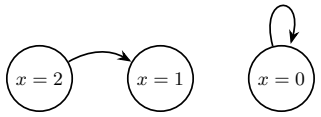


Fig. 1. Visualization of the transition system S of Example 1.

We also modify the property to check instead for a pair of $\neg q$-states not in any well-founded relation in $\mathbb{W}$:

$$\overline{P} := \left(T^{-1}(\neg C) \wedge \text{SVD} \wedge \neg \mathbb{W}(X_c, X)\right) \rightarrow q$$

Model checking $\overline{S}$ for the property $\overline{P}$ will return *Unsafe* if there is a path $\pi := s_0, \ldots, s_c, s'_c \ldots, s$ where:

1. SVD is set after s_c: $s_c \not\models \text{SVD}$, $s'_c \models \text{SVD}$, $s \models \text{SVD}$.
2. s and s_c are $\neg q$-states: $s \models \neg q$, $s_c \models \neg q$.
3. (s_c, s) are not part of any relation in $\mathbb{W}$: for all $W \in \mathbb{W}, \neg W(s_c, s)$.
4. s is not a predecessor of a shoal: $s \models T^{-1}(\neg C)$

As a result, the algorithm constructs a candidate counterexample segment-by-segment, where each segment has at least one pair of non-well-founded $\neg q$-states. Conversely, the algorithm learns states that cannot be part of any counterexample (i.e., shoals) if all pairs of $\neg q$-states are in some well-founded relation or no reachable $\neg q$-state pair exists.

Shoal Maintenance In general, inductive invariants produced by the check-inv call on line 13 might contain the extended variables SVD and X_c. Such variables are not part of the original system, and should therefore be removed from the invariants in order to produce shoals. Note however that simply applying existential quantifier elimination for this would not work, as the resulting formula could be too general. Before applying existential elimination, it is necessary to force SVD to be false, as done on line 21. The following example illustrates the situation.

Example 1. Consider the following transition system $S := \langle X, I, T \rangle$:

$$X := \{x\} \qquad I := \top \qquad T := ((x = 2) \wedge (x' = 1)) \vee ((x = 0) \wedge (x' = 0)).$$

A graphical illustration of S is shown in Figure 1. Suppose we are trying to prove $\mathbf{FG}q$, where $q := (x < 0) \vee (x > 2)$, and that $\mathbb{P} := \{(x = 0), (x = 1), (x = 2)\}$ and $\mathbb{W} := \{W\}$, where $W(y, z) := (y > z) \wedge (z \geq 0)$. Note that $S \not\models \mathbf{FG}q$, since the path in which x is always set to 0 is a counterexample for the property. However, suppose that rlive-inf picks $\widehat{s} := (x = 2)$ as the $\neg q$-state on line 5. Then, the check on line 13 will return *Safe*, and suppose now that check-inv returns the following inductive invariant:

$$\begin{aligned}\overline{inv} := &((x = 2) \wedge \neg\text{SVD}) \vee ((x = 1) \wedge (\text{SVD} \rightarrow (x_c \geq 2))) \vee \\ &(\text{SVD} \wedge (x_c = 1) \wedge (x = 0)).\end{aligned}$$

Since $\overline{inv}$ contains the auxiliary variables x_c and SVD, we cannot take it directly as a shoal, because otherwise the check-inv call on line 4 could still return the same abstract state $\widehat{s}$, since $\widehat{s} \wedge \overline{T}^{-1}(\neg\overline{inv}) \not\models \bot$. However, simply projecting away the auxiliary variables via existential quantifier elimination would be unsound, as it would generalize $\overline{inv}$ too much: since $\exists(x_c, \text{SVD}).\overline{inv}$ is equivalent to $(x = 0) \vee (x = 1) \vee (x = 2)$, adding it as a shoal would block also the state $(x = 0)$, thus preventing rlive-inf to discover the counterexample and making it wrongly conclude that the property holds. By forcing SVD to false before applying existential elimination, instead, we obtain the formula $(x = 2) \vee (x = 1)$, which is a valid shoal. ◇

Finally, we would like to remark that in a practical implementation, quantifier elimination is typically not necessary. For example, if check-inv is based on some variant of IC3 such as [13], it is possible to prove that the inductive invariants produced on line 20 are of the form $\phi(X) \wedge \bigwedge_i(\text{SVD} \rightarrow \psi_i(X, X_c))$. In such cases, the transformation of line 21 to obtain shoals amounts to simply taking $\phi(X)$.

Example 2. We conclude the section with an example illustrating the behaviour of rlive-inf. Consider the following symbolic transition system $S := \langle X, I, T\rangle$, where:

$$\begin{aligned} X &:= \{l_0, l_1, x, y\} \\ I &:= L_A \\ T &:= \big(L_A \wedge L'_B\big)\vee \\ &\quad \big(L_B \wedge L'_B \wedge (x > 0) \wedge (x' < x)\big)\vee \\ &\quad \big(L_B \wedge L'_C\big) \vee \big(L_A \wedge L'_C\big)\vee \\ &\quad \big(L_C \wedge L'_C \wedge (y < 2) \wedge (y < y')\big)\vee \\ &\quad \big(L_C \wedge L'_D\big) \vee \big(L_B \wedge L'_D\big) \vee \big(L_D \wedge L'_D\big) \end{aligned}$$

where $L_A := \neg l_0 \wedge \neg l_1$, $L_B := l_0 \wedge \neg l_1$, $L_C := \neg l_0 \wedge l_1$, $L_D := l_0 \wedge l_1$, and $x, y \in \mathbb{Z}$. We call out specifically that self-loop transitions on L_B and L_C cannot be taken forever — no matter the value of x and y initially, the self-loop guard conditions will eventually fail since at each transition x decreases in L_B and y increases in L_C. A visualization of the example is shown in Figure 2. We now outline how rlive-inf proves that $S \models \mathbf{FG}L_D$ (thus, in this example, $q = L_D$).

Initially, $\mathbb{P} := \{l_0, l_1\}$, $\mathbb{W} := \emptyset$, and $C := \bot$. The first check-inv call (line 4) will cause the abstract $\neg q$-state $\widehat{s_0} := \{\neg l_0, \neg l_1\}$ to be added to the stack $\widehat{B}$. The subsequent call to check-inv on line 13 will reach a new abstract $\neg q$-state $\widehat{s_1} := \{l_0, \neg l_1\}$, which can also be reached again in the subsequent iteration of the inner loop of line 7, resulting in the abstract lasso $\widehat{s_0}, \widehat{s_1}, \widehat{s_1}$. The counterexample is spurious, however, and by analyzing the concretization of the looping part of the abstract lasso, refinement finds the well-founded relation $W_1(x_c, x) := (x_c > x) \wedge (x \geq 1)$, together with a new predicate $(x \geq 1)$, in order to improve the precision of the abstraction. [4]

The CEGAR loop then executes a new iteration with an updated precision $\mathbb{P} := \{l_0, l_1, (x \geq 1)\}$ and $\mathbb{W} := \{W_1\}$. abs-rlive-wfr will then produce another

[4] Our refinement procedure is discussed in Section 4.1. We refer to Section 4.2 of [23] for additional details.

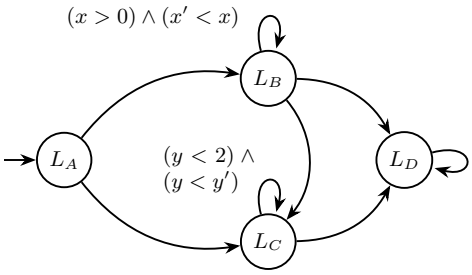


Fig. 2. Visualization of the transition system S of Example 2 with edges labeled with constraints placed on x and y wrt L_A, L_B, L_C, and L_D.

abstract lasso, namely: $\{\neg l_0, \neg l_1, (x \geq 1)\}$, $\{\neg l_0, l_1, (x \geq 1)\}$, $\{\neg l_0, l_1, (x \geq 1)\}$. Also this counterexample is spurious, and the abstraction in this case can be refined by adding the relation $W_2(y_c, y) := (y > y_c) \wedge (y \leq 1)$ to $\mathbb{W}$ and the predicate $(y \leq 1)$ to $\mathbb{P}$. At this point, the abstraction is sufficiently precise to make abs-rlive-wfr return *Safe*. More specifically:

- The check-inv call on line 4 adds the abstract state $\widehat{s_0} := \{l_0, \neg l_1, \neg(x \geq 1), \neg(y \leq 1)\}$ to the stack $\widehat{B}$;
- The check-inv call on line 13 finds a segment starting from $\widehat{s_0}$ to the abstract state $\widehat{s_1} := \{\neg l_0, l_1, \neg(x \geq 1), (y \leq 1)\}$, which is then added to $\widehat{B}$;
- The next check-inv call, to find another segment starting from $\widehat{s_1}$ and reaching the next $\neg q$-state, finds $\widehat{s_2} := \{\neg l_0, l_1, \neg(x \geq 1), \neg(y \leq 1)\}$ and adds it to $\widehat{B}$;
- At this point, the search for another segment starting from $\widehat{s_2}$ is unsuccessful, since all its successors are q-states. Therefore, the check-inv call on the extended system $\langle \overline{X}, [\![s_2]\!] \wedge \neg \text{SVD}, \overline{T} \rangle$ and invariant property $\overline{P}$ returns *Safe* and produces the inductive invariant $l_1 \wedge (l_0 \vee \neg(y \leq 1)) \wedge (\text{SVD} \rightarrow l_0)$, from which the shoal $l_1 \wedge (l_0 \vee \neg(y \leq 1))$ is extracted on line 21 and added to C.
- $\widehat{s_2}$ is then removed from $\widehat{B}$, and another call to check-inv is performed to find another abstract lasso segment starting from $\widehat{s_1}$. Also in this case, however, the search is not successful. This is because all successors of $\widehat{s_2}$ in S that satisfy $\neg q$ are either satisfying W_2, or contained in C; therefore, check-inv returns *Safe* and generates the invariant $l_1 \wedge \neg l_0 \wedge (\text{SVD} \rightarrow (y < y_c))$, resulting in the new shoal $l_1 \wedge \neg l_0$. C is then updated to $(l_1 \wedge (l_0 \vee \neg(y \leq 1))) \vee (l_1 \wedge \neg l_0)$.
- $\widehat{s_1}$ then is removed from $\widehat{B}$, and check-inv is called again on line 13 to find a segment starting from $\widehat{s_0}$. The seach is again unsuccessful, with check-inv returning *Safe* with the invariant $l_0 \wedge (\text{SVD} \rightarrow (x_c > x))$, resulting in the shoal l_0, thus updating C to $(l_1 \wedge (l_0 \vee \neg(y \leq 1))) \vee (l_1 \wedge \neg l_0) \vee l_0$. In this case, the use of W_1 is crucial for excluding $\neg q$-successors.
- $\widehat{s_0}$ is removed from $\widehat{B}$, which becomes empty. A new iteration of the outer loop calls check-inv on line 4, finding no more $\neg q$-states outside the shoals, and rlive-inf returns *Safe*. ◇

5 Correctness

In this section, we prove that rlive-inf does not produce wrong results. We begin by proving the correctness of Algorithm 3, namely that it either proves that $S \models \mathbf{FG}q$ or it finds an abstract lasso. Note that if there is there is no abstract lasso, then $S \models \mathbf{FG}q$. This follows immediately from the following lemma proved in [23].

Lemma 1 (Lemma 1 of [23]). *Let π be a path satisfying $\boldsymbol{GF}\neg q$ and $\mathbb{W}$ be a finite set of well-founded relations. Then, any infinite suffix π' of π contains two states s_1, s_2, each satisfying $\neg q$, such that (s_1, s_2) is not in any relation in $\mathbb{W}$.*

Corollary 1. *If S has a path π satisfying $\boldsymbol{GF}\neg q$, then π is an abstract lasso for any sets $\mathbb{P}$ and $\mathbb{W}$.*

The main non-trivial point for proving the correctness of Algorithm 3 is the shoal construction defined at line 21. In the following lemma, we prove that there is no abstract lasso starting from any state in $C = \exists X_c, \text{SVD}.\ \big(\overline{inv} \wedge \neg\text{SVD}\big)$.

Lemma 2 (Shoal Correctness). *Let a variable set X, initial condition I, transition relation T, transition system $S := \langle X, I, T\rangle$, and a formula $C(X)$ such that there is no abstract lasso starting from C in S be given. Let $\overline{S} := \langle \overline{X}, [\![\widehat{s}]\!] \wedge \neg\text{SVD}, \overline{T} \wedge (\neg C \wedge \neg C')\rangle$ where $\widehat{s}$, $\overline{X}$ and $\overline{T}$ are defined as in lines 8, 10 and 11 of Algorithm 3. Let $\overline{inv}$ be an inductive invariant entailing $\neg\overline{P}$ as defined in line 12. Then there is no abstract lasso starting from $C_{new} = \exists X_c, \text{SVD}.\ \big(\overline{inv} \wedge \neg\text{SVD}\big)$.*

Proof. Suppose by contradiction that there is an abstract lasso $\pi := s_0, s_1, \ldots$ starting from a state s_0 in C_{new}. By definition, there exist s_i, s_j, s_h, and s_k (with $i \leq h < k \leq j$) such that:

1. $s_i \models \neg q$, $s_j \models \neg q$, $s_h \models \neg q$, $s_k \models \neg q$;
2. for all $w \in \mathbb{W}$, $(s_h, s_k) \notin w$.

We can extend π to a path $\overline{\pi}$ over $\overline{X}$ by setting SVD to false in $\overline{s}_0, \ldots, \overline{s}_h$, and true elsewhere, and by setting X_c to the value of X in s_h everywhere. In this way, for all $l \geq 0$, $\overline{s}_l, \overline{s}_{l+1} \models \overline{T}$. Since $\overline{inv}$ is inductive, then for all $l \geq 0$, $\overline{s}_l \models \overline{inv}$. However, $\overline{s}_k \models \neg q$ as mentioned before, $\overline{s}_k \models \text{SVD}$ by construction, $\overline{s}_k \models \neg\mathbb{W}(X, X_c)$ because $s_h, s_k \models \neg\mathbb{W}$. Finally, note that s_{h+1} cannot be in C. Thus, we reach a contradiction with the fact that $\overline{inv}$ entails $\neg\overline{P}$. □

Theorem 1 (abs-rlive-wfr Soundness). *Assuming that check-inv is correct, if Algorithm 3 returns Safe, then $S \models \boldsymbol{FG}q$; if Algorithm 3 returns Unsafe, then S has an abstract lasso.*

Proof. The algorithm returns *Safe* when check-inv$(X, I, T \wedge (\neg C \wedge \neg C'), \neg q \wedge T^{-1}(\neg C))$ returns *Safe*. Thus no state in $\neg C$ can reach $\neg q$. Note that C is updated only at Line 21 and by Lemma 2, the algorithm maintains as invariant that C always contains states that cannot be the starting point of an abstract lasso. Thus, by Lemma 1, $S \models \mathbf{FG}q$.

The algorithm returns *Unsafe* when a new state $\widehat{t}$ is found as the last state of a counterexample by $\mathsf{check\text{-}inv}(\overline{X}, [\![\widehat{s}]\!] \wedge \neg \mathrm{SVD}, \overline{T} \wedge (\neg C \wedge \neg C'), \overline{P})$ and $\widehat{t}$ is already on the stack of abstract states. For every pair of abstract states $\widehat{s}$ and $\widehat{t}$ that are consecutive on the stack, there exist a path $\pi := s_0, s_1, \ldots, s_k$ such that $s_0 \in [\![\widehat{s}]\!]$ and $s_k \in [\![\widehat{t}]\!]$; and for some h, for all $W \in \mathbb{W}$, $(s_h, s_k) \notin W$. Moreover there exists a path $\pi := s_0, s_1, \ldots, s_k$ where $s_0 \models I$ and the abstract state of s_k is the first on the stack. Finally, all abstract states on the stack satisfy $\neg q$. Therefore, there exists an abstract lasso. □

Theorem 1 allows us to prove the correctness of rlive-inf, provided that the procedures feasible and refine of Algorithm 2 satisfy some basic soundness assumptions. Specifically, we require feasible to return true only if there exists an infinite path in S in which $\neg q$ is true infinitely often (and that is consistent with the abstract lasso), and refine to only produce relations that are well-founded. We can then state the following.

Theorem 2 (rlive-inf Soundness). *Assuming that feasible and refine are correct, if Algorithm 2 returns Safe, then $S \models \boldsymbol{FG}q$, and if it returns Unsafe, then $S \not\models \boldsymbol{FG}q$.*

6 Limitations

Theorem 2 ensures that rlive-inf is sound, i.e. that a *Safe* result returned by Algorithm 2 implies that $S \models \mathbf{FG}q$, and an *Unsafe* one implies that $S \not\models \mathbf{FG}q$. However, since liveness checking for infinite-state systems is in general an undecidable problem, rlive-inf is necessarily incomplete.

A first source of incompleteness lies in the fact that rlive-inf currently only returns *Unsafe* if it finds a lasso-shaped counterexample, and therefore it will diverge (i.e., not terminate) for even simple systems whose only counterexamples are not lasso-shaped. This limitation can however be addressed by integrating techniques for LTL falsification in infinite-state systems such as [10]. A second source of incompleteness is that the calls to check-inv in abs-rlive-wfr are in general undecidable. Third, both interpolation-based refinement and ranking function synthesis procedures used for abstraction refinement are incomplete and might either fail to refine the precision of the abstraction, or produce an infinite sequence of refinements.

Finally, and perhaps more interestingly, another source of incompleteness is inherent to the way well-founded relations are integrated in the rlive search. The extended system encoding presented in §4.2, justified by Lemma 1, implies that abs-rlive-wfr returns an abstract lasso as soon as it finds *any* pair of $\neg q$-states that are not in any relation in $\mathbb{W}$. However, if that pair is part of a path that necessarily visits a different pair of abstract states such that all their concretizations are in some relation already in $\mathbb{W}$, then the entire abstract lasso is spurious. The following is an example illustrating such situation. Addressing such limitation of rlive-inf (as well as assessing its impact in practice) is part of our plans for future work.

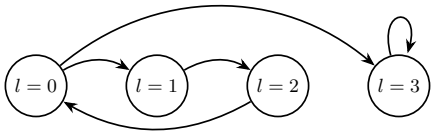


Fig. 3. Visualization of the transition system S of Example 3.

Example 3. Consider the following system $S := \langle X, I, T \rangle$ where:

$$
\begin{aligned}
X &:= \{l, x, q\} \\
I &:= (l = 0) \wedge (x > 0) \wedge q \\
T &:= \big((l = 0) \wedge (l' = 1) \wedge (x > 0) \wedge (x' = x - 1) \wedge \neg q'\big) \vee \\
&\quad \big((l = 1) \wedge (l' = 2) \wedge (x' = x + 1) \wedge \neg q'\big) \vee \\
&\quad \big((l = 2) \wedge (l' = 0) \wedge (x' = x - 1) \wedge \neg q'\big) \vee \\
&\quad \big((l = 0) \wedge (l' = 3) \wedge (x \leq 0) \wedge q'\big) \vee \\
&\quad \big((l = 3) \wedge (l' = 3) \wedge q'\big)
\end{aligned}
$$

shown graphically in Figure 3. The system satisfies the property $\mathbf{FG}q$, because every loop from $l = 0$ to $l = 2$ (in which $\neg q$ holds) can only be executed a finite number of times, since the transition from $l = 0$ to $l = 1$ is guarded by a condition $(x > 0)$, and inside the loop x is decremented twice and only incremented once. Therefore, a suitable well-founded relation that proves that the loop cannot be executed infinitely often is $W(x_c, x) := (x_c > x) \wedge (x \geq 0)$. However, since the transition from $l = 1$ to $l = 2$ violates W, abs-rlive-wfr will find an abstract lasso and rlive-inf won't be able to prove the property. ◇

7 Related Work

Modern symbolic LTL model checking algorithms predominantly rely on SAT-based methods. Some of them can be trivially extended to infinite-state systems replacing SAT with SMT solvers.

For example, rlive and k-liveness [16] remain sound also for infinite-state systems, as they are based on proving $\mathbf{FG}q$ properties by showing that $\neg q$ can be visited only a finite number of times. k-liveness focuses on counting the number of times $\neg q$ can be visited, while rlive looks for counterexamples by state enumeration, while blocking states that can reach $\neg q$ only a bounded number of times (the "shoals").

Both rlive and k-liveness rely on invariant model checking with a series of reachability checks trying to reach $\neg q$. These subproblems can be addressed with implicit predicate abstraction [44] as done for example in [12] for k-liveness. However, they focus on finite paths, and are not amenable to include proofs based on well-founded relations. We here addressed this challenge in the case of rlive, but the same idea can be in principle extended to k-liveness.

FAIR [8] and liveness-to-safety [3] search only for lassos and are therefore unsound when extending to the infinite-state case by just replacing the SAT

solver with an SMT one. The liveness-to-safety transformation was adapted for infinite-state systems in [23] with implicit predicate abstraction and well-founded relations. This approach was shown to be also more effective than algorithms focused on termination, also employing well-founded relations (cfr.,e.g., [24] and [42]). The extension of rlive is similar to the one presented in [23]. As we discussed above, it uses the same high-level CEGAR loop and the same refinement of predicates and well-founded relations. αL2S also shares the same limitations described in Section 6 and in particular cannot prove Example 3. However, the details of rlive-inf are quite different from αL2S. In particular, αL2S uses a monolithic encoding to find candidate counterexamples, using a single invariant model-checking call wrt. the current level of abstraction, whereas rlive-inf builds counterexamples iteratively with local searches that do not consider loop conditions. Further, rlive-inf explicitly caches states known to visit the fairness condition a finite number of times in shoals. While the formulation of αL2S does not do any caching, in practice the implementation of [23] uses certain invariants found from previous calls to the underlying ic3-based model checker.

Other techniques such as [30,22,41,45,27] focus specifically on proving termination of programs, sometimes under fairness conditions [19]. Extensions of these techniques to proving temporal properties of systems include [21,20], though they do not focus on liveness checking of general symbolic transition systems.

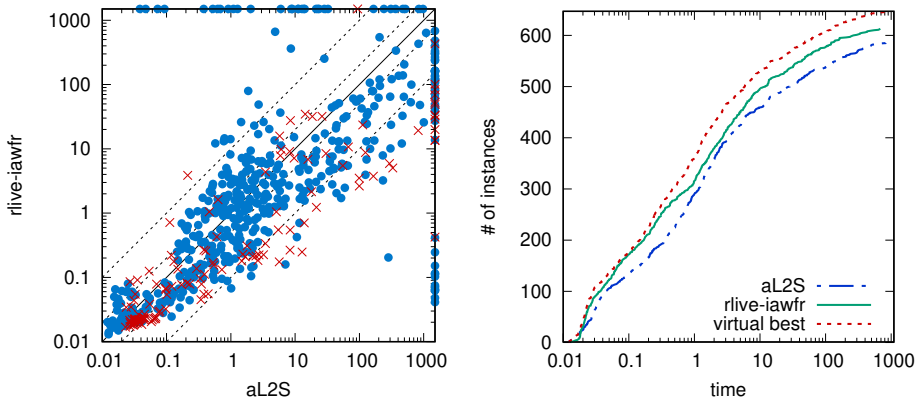
Another recent approach to LTL model checking uses neural networks to search for ranking functions that prove that there is no fair trace in the system satisfying the property [26]. The technique is based on generating candidate fair termination certificates by training a neural network on random traces of the system, and then checking the certificates with an SMT solver. When the check fails, the SMT counterexample is used for retraining and generate another candidate certificate. The approach is aimed at hardware designs, although in principle it could be generalised to infinite-state systems as well.

The model checking approach presented in [25] provides a BMC encoding for checking general LTL properties on infinite-state systems. The technique provides certain completeness guarantees given a finite domain and considers a variety of trace semantics beyond just infinite traces, though the BMC encoding especially limits its ability to prove properties and will diverge on many problem instances with infinite domains. Our technique uses implicit abstraction to deal with the challenges of infinite domains.

8 Experimental Evaluation

In this section, we experimentally evaluate the performance of rlive-inf. We have implemented the algorithm on top of ic3ia [34], an open source model checker based on IC3 with implicit abstraction [13], written in C++. [5] Although the rlive-inf procedure is independent from the underlying theory used to specify

[5] An artifact enabling full reproducibility, including also the source code of our implementation and log files of our experiments, is available at `https://doi.org/10.5281/zenodo.15310253`



Algorithm	# Solved	Safe	Unsafe	$\Delta_{\text{rlive-inf}}$	Gained	Lost	Cumulative time (sec)
Virtual best	647	513	134	34	34	0	15164
rlive-inf	613	480	133	–	–	–	13110
αL2S	591	470	121	-22	34	56	22115

Fig. 4. Experimental results.

the input system, our implementation currently targets systems expressed using Boolean and linear arithmetic constraints.

Experimental set up We compare rlive-inf to αL2S, which was shown in [23] to outperform other approaches to LTL verification for infinite-state symbolic transition systems, and which to the best of our knowledge still represents the state of the art in this context. Moreover, since αL2S was also implemented on top of ic3ia, we reused the same basic components (namely, invariant verification engine [13], SMT solver [14], predicate refinement procedure [32] and generation of well-founded relations via ranking function synthesis techniques [31]) for implementing rlive-inf. This ensures that the comparison between the two approaches is fair, and any differences in performance is due to the different features and search strategies employed by the two techniques, rather than to the different performance of the underlying engines.

Benchmarks We use the benchmark set of the αL2S article [23] for our comparison. The set consists of 835 liveness verification problems expressed as symbolic transition systems in the VMT format [15], coming from multiple sources (including BIP models from [6], examples derived from the real-time domain [1], imperative programs from [24], and termination benchmarks from [9]).

Results We ran our experimental evaluation on a cluster of machines with AMD EPYC 7413 CPUs and 500Gb of RAM, running Ubuntu Linux 20.04. We used a timeout of 1200 seconds and a memory limit of 8Gb per instance.

The results of the evaluation are summarized in Figure 4. The scatter plot on the left shows a direct comparison of the run times of rlive-inf and αL2S on each individual instance (where safe ones are shown as blue dots and unsafe ones as red crosses), whereas the plot on the right shows, for each tool, the number of solved instances (y-axis) in the given time time (x-axis), not including timeouts/unknowns. In this case, we also include a "virtual best" tool, obtained by taking the best among rlive-inf and αL2S on each instance. Additional information is provided in the table under the plots, where for each tool we show the number of solved instances (distinguishing also between safe and unsafe ones), the difference in number of solved instances wrt. rlive-inf, the number of instances gained (i.e. solved by the given tool but not by rlive-inf) and lost, and the total execution time taken on solved instances. In all cases, both algorithms agreed on their answers and no memouts occurred.

From the results, we can conclude that rlive-inf outperforms αL2S both in terms of number of solved instances (solving 22 more instances than αL2S) and in terms of runtime efficiency (with an average speedup of 6.47x on instances solved by both tools). From the plot on the right, we can see that the efficiency advantage remains regardless of the selected time limit (within the overall timeout of 1200 seconds used to run the experiments), shown by the fact that the rlive-inf curve always dominates the one corresponding to αL2S. Interestingly however, the two techniques compared show a high degree of complementarity, as highlighted by the scatter plot on the left and by the strong results obtained by the "virtual best" configuration both in terms of number of solved instances (34 more than rlive-inf alone) and of runtime efficiency. We believe that our results demonstrate the practical contributions of our new procedure to advancing the state of the art in LTL verification for infinite-state systems.

9 Conclusions and Future Work

We have presented rlive-inf, an generalisation of the rlive liveness model checking algorithm from finite- to infinite-state symbolic transition systems. By integrating predicate abstraction, invariant checking, and termination techniques based on well-founded relations, rlive-inf provides an efficient, fully-symbolic LTL model checker that outperforms the state of the art on a variety of benchmarks.

Regarding future work, we intend to address the limitation described in §6 concerning the integration of well-founded relations in the rlive search. We will also investigate the use of improved strategies for refinement, leveraging additional techniques for raking function synthesis such as [45], since this is often a limiting factor in the effectiveness of the approach. Finally, we intend to integrate techniques for the discovery of non-looping counterexamples for violated properties [10], and to study the problem of generating proof certificates for correctness.

Acknowledgements. A. Cimatti, A. Griggio and S. Tonetta have been supported by the PNRR project FAIR - Future AI Research (PE00000013), under the NRRP MUR program funded by the NextGenerationEU and the PNRR MUR project VITALITY (ECS00000041), Spoke 2 ASTRA - Advanced Space Technologies and Research Alliance. C. Johannsen and K.Y. Rozier have been supported by NSF:CCRI Award #2016592.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Rajeev Alur, Thao Dang, and Franjo Ivancic. Counterexample-guided predicate abstraction of hybrid systems. *Theor. Comput. Sci.*, 354(2):250–271, 2006.
2. Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 1267–1329. IOS Press, 2021.
3. A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. In *Proc. 7th Int. Workshop on Formal Methods for Industrial Critical Systems*, volume 66:2 of *Electronic Notes in Theoretical Computuer Science*, 2002.
4. Johannes Birgmeier, Aaron R Bradley, and Georg Weissenbacher. Counterexample to induction-guided abstraction-refinement (CTIGAR). In *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26*, pages 831–848. Springer, 2014.
5. Nikolaj S. Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation II*, volume 9300 of *Lecture Notes in Computer Science*, pages 24–51. Springer, 2015.
6. Simon Bliudze, Alessandro Cimatti, Mohamad Jaber, Sergio Mover, Marco Roveri, Wajeb Saab, and Qiang Wang. Formal Verification of Infinite-State BIP Models. In *ATVA*, volume 9364 of *LNCS*, pages 326–343. Springer, 2015.
7. Alberto Bombardelli, Alessandro Cimatti, Alberto Griggio, and Stefano Tonetta. Another Look at LTL Modulo Theory over Finite and Infinite Traces. In *Principles of Verification (1)*, volume 15260 of *Lecture Notes in Computer Science*, pages 419–443. Springer, 2024.
8. Aaron R Bradley, Fabio Somenzi, Zyad Hassan, and Yan Zhang. An incremental approach to model checking progress properties. In *2011 Formal Methods in Computer-Aided Design (FMCAD)*, pages 144–153. IEEE, 2011.
9. Marc Brockschmidt, Byron Cook, and Carsten Fuhs. Better termination proving through cooperation. In *CAV*, volume 8044 of *LNCS*, pages 413–429. Springer, 2013.
10. Alessandro Cimatti, Alberto Griggio, and Enrico Magnago. LTL falsification in infinite-state systems. *Inf. Comput.*, 289(Part):104977, 2022.
11. Alessandro Cimatti, Alberto Griggio, Sergio Mover, Marco Roveri, and Stefano Tonetta. Verification modulo theories. *Formal Methods in System Design*, 60(3):452–481, 2022.

12. Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. Verifying LTL properties of hybrid systems with k-liveness. In *CAV*, volume 8559 of *Lecture Notes in Computer Science*, pages 424–440. Springer, 2014.
13. Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. Infinite-state invariant checking with IC3 and predicate abstraction. *Formal Methods in System Design*, 49:190–218, 2016.
14. Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In *TACAS*, volume 7795 of *LNCS*. Springer, 2013.
15. Alessandro Cimatti, Alberto Griggio, and Stefano Tonetta. The VMT-LIB language and tools. In *SMT*, volume 3185 of *CEUR Workshop Proceedings*, pages 80–89. CEUR-WS.org, 2022.
16. Koen Claessen and Niklas Sörensson. A liveness checking algorithm that counts. In *2012 Formal Methods in Computer-Aided Design (FMCAD)*, pages 52–59. IEEE, 2012.
17. Edmund Clarke, Orna Grumberg, and Kiyoharu Hamaguchi. Another look at LTL model checking. In *Computer Aided Verification: 6th International Conference, CAV'94 Stanford, California, USA, June 21–23, 1994 Proceedings 6*, pages 415–427. Springer, 1994.
18. Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
19. Byron Cook, Alexey Gotsman, Andreas Podelski, Andrey Rybalchenko, and Moshe Y Vardi. Proving that programs eventually do something good. *ACM SIGPLAN Notices*, 42(1):265–276, 2007.
20. Byron Cook, Heidy Khlaaf, and Nir Piterman. On automation of CTL* verification for infinite-state systems. In *International Conference on Computer Aided Verification*, pages 13–29. Springer, 2015.
21. Byron Cook, Eric Koskinen, and Moshe Vardi. Temporal property verification as a program analysis task. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, pages 333–348. Springer, 2011.
22. Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. *ACM Sigplan Notices*, 41(6):415–426, 2006.
23. Jakub Daniel, Alessandro Cimatti, Alberto Griggio, Stefano Tonetta, and Sergio Mover. Infinite-state liveness-to-safety via implicit abstraction and well-founded relations. In *CAV (1)*, volume 9779 of *Lecture Notes in Computer Science*, pages 271–291. Springer, 2016.
24. Daniel Dietsch, Matthias Heizmann, Vincent Langenfeld, and Andreas Podelski. Fairness Modulo Theory: A New Approach to LTL Software Model Checking. In *CAV (1)*, volume 9206 of *LNCS*, pages 49–66. Springer, 2015.
25. David Doose and Julien Brunel. Simple LTL model checking on finite and infinite traces over concrete domains. In *International Conference on Formal Engineering Methods*, pages 375–390. Springer, 2024.
26. Mirco Giacobbe, Daniel Kroening, Abhinandan Pal, and Michael Tautschnig. Neural model checking. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*. NeurIPS, 2024.
27. Mirco Giacobbe, Daniel Kroening, and Julian Parsert. Neural termination analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 633–645, 2022.
28. S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In *CAV*, pages 72–83, 1997.

29. Ákos HAJDU, Tamás TÓTH, and András VÖRÖS. A survey on CEGAR-based model checking. Master's thesis, Budapest University of Technology and Economics, 2015.
30. Matthias Heizmann, Jochen Hoenicke, Jan Leike, and Andreas Podelski. Linear ranking for linear lasso programs. In *Automated Technology for Verification and Analysis: 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings*, pages 365–380. Springer, 2013.
31. Matthias Heizmann, Jochen Hoenicke, Jan Leike, and Andreas Podelski. Linear Ranking for Linear Lasso Programs. In *ATVA*, volume 8172 of *LNCS*, pages 365–380. Springer, 2013.
32. T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. Abstractions from Proofs. In *POPL*, pages 232–244, 2004.
33. Kryštof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 157–171. Springer, 2012.
34. `ic3ia` webpage. `https://es-static.fbk.eu/people/griggio/ic3ia/`.
35. Alexander Ivrii, Ziv Nevo, and Jason Baumgartner. k-FAIR= k-LIVENESS+ FAIR revisiting SAT-based liveness algorithms. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–5. IEEE, 2018.
36. Ranjit Jhala, Andreas Podelski, and Andrey Rybalchenko. Predicate abstraction for program verification. In *Handbook of Model Checking*, pages 447–491. Springer, 2018.
37. Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. SMT-based model checking for recursive programs. *Formal Methods in System Design*, 48:175–205, 2016.
38. Kenneth L. McMillan. Interpolation and model checking. In *Handbook of Model Checking*, pages 421–446. Springer, 2018.
39. Aina Niemetz, Mathias Preiner, and Armin Biere. Turbo-charging lemmas on demand with don't care reasoning. In *2014 Formal Methods in Computer-Aided Design (FMCAD)*, pages 179–186. IEEE, 2014.
40. Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 239–251. Springer, 2004.
41. Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004.*, pages 32–41. IEEE, 2004.
42. Andreas Podelski and Andrey Rybalchenko. Transition predicate abstraction and fair termination. *ACM Trans. Program. Lang. Syst.*, 29(3):15–es, May 2007.
43. Tobias Schuele and Klaus Schneider. Bounded model checking of infinite state systems. *Formal Methods in System Design*, 30:51–81, 2007.
44. Stefano Tonetta. Abstract model checking without computing the abstraction. In *FM*, volume 5850 of *Lecture Notes in Computer Science*, pages 89–105. Springer, 2009.
45. Caterina Urban, Arie Gurfinkel, and Temesghen Kahsai. Synthesizing ranking functions from bits and pieces. In *TACAS*, volume 9636 of *Lecture Notes in Computer Science*, pages 54–70. Springer, 2016.
46. Moshe Y Vardi. An automata-theoretic approach to linear temporal logic. *Logics for concurrency: structure versus automata*, pages 238–266, 2005.
47. Yechuan Xia, Alessandro Cimatti, Alberto Griggio, and Jianwen Li. Avoiding the shoals - a new approach to liveness checking. In *International Conference on Computer Aided Verification*, pages 234–254. Springer, 2024.