Certifying rlive: a New Proof Strategy for Liveness Model Checking

Giulia Sindoni[0000−0003−4003−2317], Alberto Griggio[0000−0002−3311−0893], and Stefano Tonetta[0000−0001−9091−7899]

Fondazione Bruno Kessler, Trento, IT
{gsindoni,griggio,tonettas}@fbk.eu

Abstract. SAT-based model checking has become a prominent approach to the verification of temporal properties. However, while invariant model checking can produce simple proofs based on induction, proof generation for SAT-based model checking of liveness properties is much more complex.
In this paper, we focus on a recently developed algorithm, called rlive, which has been proved quite effective in practice. rlive tries to find a counterexample with a series of reachability checks, while iteratively blocking shoals, i.e., set of states that cannot be extended with fair paths. Despite the complexity of the algorithm, we show that the shoals are sufficient to generate a proof in a deductive system for temporal properties. We implement the approach in an existing certifying model checking framework based on the PVS theorem prover, and we experimentally evaluate it on liveness verification problems from the hardware model checking competition, generating proofs using the nuXmv model checker and checking them with PVS.

1 Introduction

Applying formal methods as a tool for certifying high-assurance and safety-critical systems demands for verification tools that are capable of providing a high level of confidence in their outcomes.

A model checker generally offers a straightforward "yes" or "no" answer when addressing a verification problem. When the answer is "no" the model checker provides a counterexample as supporting evidence. No such evidence is usually given when the answer is "yes". Moreover, the growing complexity of model checkers themselves has made it increasingly important to obtain certificates from their process. The idea of certifying model checking [19] is to generate certificates as a byproduct of the verification. These certificates, often in the form of deductive proofs, serve to build trust in the verification results by providing additional evidence of correctness.

In this paper we consider problem of certifying the liveness checking problem, denoted $\mathcal{M} \models \mathbf{FG}q$ where $\mathbf{FG}q$ intuitively means that, in any satisfying trace, q eventually holds in all the future states. More specifically, we focus on rlive [30],

a new SAT-based model-checking algorithm for the verification of liveness properties of finite-state symbolic transition systems, which has been proved quite effective in practice. rlive tries to find a counterexample with a series of reachability checks, while iteratively blocking a set of shoals, i.e. set of states that cannot reach a $\neg q$-state infinitely often.

We show that, despite the complexity of this algorithm, the shoals are sufficient to generate certifying proofs for the liveness property in a deductive system. The shoals are easily provided by the model checker, in the case where this has verified that the model satisfies the liveness property.

Our starting point is the approach presented in [15], describing a method for the generation of liveness checking certificates using the k-liveneness algorithm [9]. The key idea of k-liveness is it prove that a liveness property holds, by bounding how many times q can be false. For any valid liveness property in a finite-state transition system, there exists a bound k such that q can become false at most k times in any trace. The algorithm incrementally searches for this bound ($k = 0, 1, 2, \ldots$), using a SAT-based safety checker to verify each bound until one succeeds. In this paper, we adapt and generalise the certification procedure for k-liveness of [15] to make it applicable also to rlive. We then extend our previous work of [28], where we presented a theorem prover based certification framework for invariant properties, to handle liveness proofs.

More specifically, we make the following contributions:

1. The formalisation of new temporal deductive rule capturing rlive. We formally prove the correctness and completeness of the rule. This rule is a generalisation of the temporal rule for k-liveness presented in [15]. The correctness result is proved within a deductive proof system developed in the PVS specification language [26], so that it can be used within an automated proof strategy.

2. The development of a proof strategy for certifying the success of the model checking answer "yes", when the model checker has used rlive to show that the model satisfies the liveness property. This strategy only takes as input the set of shoals, which are created during the model checking stage.

We experimentally evaluate our approach on liveness verification problems from the hardware model checking competition, generating proofs using the nuXmv model checker [7] and checking them with PVS.

Outline The rest of the paper is organised as follows. §2 provides an overview of related work. §3 introduces notation and background notions. §4 describes our contribution at formalising rlive as a temporal deductive rule and at generating proofs for certifying liveness checking results using rlive. §5 reports an evaluation of the prototype implementation of the proof strategy on a standard set of benchmarks for liveness checking. §6 concludes the paper.

2 Related Work

This work builds upon the concepts first presented in [15], where it was shown how to exploit the k-liveness algorithm to extend proof generation capabilities

for invariant checking to cover full linear-temporal logic (LTL) properties, with little overhead for the model checker. The work shows how k-liveness can be formalised as a temporal deductive rule and its correctness proved within a complete axiomatic system for LTL from [13]. However, in that case no theorem prover was used to check the proof generated by the model checker.

In [28] the authors fill this gap by presenting a novel approach for certifying model checking results exploiting a theorem prover, namely PVS, and a theory of temporal deductive rules that can support various kinds of transformations and simplifications of the original model, such as 2-phase abstraction and temporal decomposition. This work was however restricted in that it can only handle the proof of invariant properties.

The present work builds upon the theorem-prover based approach from [28], by showing how to certify model checking results for liveness properties, specifically when the model checker uses the newly introduced rlive algorithm to prove the property at issue. Even though rlive looks as a very different algorithm from k-liveness, the proof of the first turns out to be a generalisation of the latter presented in [15].

A different approach to certification is presented in [31], where a formal framework designed to certify model checking results based on k-induction is described. The core of this certification process is the creation of a witness circuit which simulates the original circuit, and which includes an inductive invariant that serves as a proof certificate. This approach is extended further in [4] and [12]. The framework is however limited to the certification of invariants, and doesn't consider liveness or general LTL properties.

A relevant approach to certifying the correctness of liveness properties specifically is presented in [16]. The authors present a variant of the liveness-to-safety algorithm [1], and transform the liveness property into a safety property using a reduction. Then they get a proof for that safety property. However, this reduction has to be trusted as correct, and the proof does not target the original system, but the result of the reduction. In our work, we produce a temporal proof of the fact that the original system satisfies the property, so that only the theorem prover performing the proof, and not the model checking algorithms used, has to be trusted.

Other approaches concerning the generation of proofs from model checking results include [23], [22], [17] and [10], but they are mainly theoretical, and no implementation is available, to the best of our knowledge. In contrast, our certification procedure has been implemented within a theorem prover framework, and specifically targets the rlive algorithm.

3 Background and Preliminaries

We operate within the framework of Boolean (propositional) logic, using the standard concepts of satisfiability, validity, interpretations, and models. We use lowercase Latin letters x, v (possibly with subscripts or primes) to denote propositional variables. Similarly, uppercase Latin letters X, V represent sets of vari-

ables. Uppercase Latin letters $\mathcal{I}$, T, as well as lowercase Latin and Greek letters q, ϕ, ψ, are used to denote formulae, while uppercase Greek letters Γ and Δ and Π represent sets of formulae.

3.1 Transitions Systems

We take into account systems modeled by state transition structures, implicitly represented by propositional formulae. A *transition system* $\mathcal{M}$ is a triple $\langle X, \mathcal{I}, T \rangle$, where X is a set of (propositional) *state variables*, $\mathcal{I}(X)$ is a formula representing the *initial states*, and $T(X, X')$ is a formula representing the system's *transition relation*. The states of $\mathcal{M}$ are (complete) assignments to the variables in X. We denote by Σ_X the set of states. A state $s \in \Sigma_X$ is a model for a propositional formula ψ, denote by $s \models \psi(X)$, if substituting the values of s into ψ, the formula ψ evaluate to *True*. Next states, i.e., those reached after a transition, are represented as assignments to primed state variables X'. A *path* of $\mathcal{M}$ is an infinite sequence of states $s_0, s_1, \ldots$ such that $s_0 \models \mathcal{I}$, and for all $i \geq 0$, $s_i, s'_{i+1} \models T$. Given a path $\pi := s_0, s_1, \ldots$ we denote with $\pi[i]$ the state s_i.

3.2 Linear Temporal Logic

Linear Temporal Logic (LTL) was introduced by Pnueli [24] for the specification and verification of reactive systems. Formulae of LTL are constructed from a set of *propositional variables* X using the usual *logical connectives* ($\neg$, $\wedge$, $\vee$) and some *temporal operators* $\mathbf{X}$ ("next"), $\mathbf{F}$ ("eventually"), $\mathbf{G}$ ("always") and $\mathbf{U}$ ("until"). LTL formulae are interpreted in terms of paths, i.e., sequences of states of a transition system. Their semantics is also extended to states and whole transition systems.

Given a transition system $\mathcal{M} = \langle X, \mathcal{I}, T \rangle$, a path $\pi := s_0, s_1, \ldots$ in $\mathcal{M}$, an index i and a formula ψ over X, we define $\pi, i \models \psi$, i.e. that π satisfies ψ in i, as follows:

- $\pi, i \models \top$ and $\pi, i \not\models \bot$.
- For each $p \in X$, $\pi, i \models p$ iff $\pi[i] \models p$.
- $\pi, i \models \neg\psi$ iff $\pi, i \not\models \psi$.
- $\pi, i \models \psi_1 \wedge \psi_2$ iff $\pi, i \models \psi_1$ and $\pi, i \models \psi_2$.
- $\pi, i \models \psi_1 \vee \psi_2$ iff $\pi, i \models \psi_1$ or $\pi \models \psi_2$.
- $\pi, i \models \mathbf{X}\psi$ iff $\pi, i+1 \models \psi$.
- $\pi, i \models \mathbf{F}\psi$ iff $\pi, j \models \psi$ for some $j \geq i$.
- $\pi, i \models \mathbf{G}\psi$ iff $\pi, j \models \psi$ for every $j \geq i$.
- $\pi, i \models \psi_1 \mathbf{U} \psi_2$ iff $\pi, j \models \psi_2$ for some $j \geq i$ and $\pi, k \models \psi_1$ for every $i \leq k < j$.

Finally, $\pi \models \psi$ iff $\pi, 0 \models \psi$. Given a propositional formula q over X, we call the *liveness checking problem*, denoted by $\mathcal{M} \models \mathbf{FG}q$, the problem to check $\pi \models \mathbf{FG}q$ for all paths π of $\mathcal{M}$. $\mathbf{FG}q$ intuitively means that, in any path of $\mathcal{M}$, q eventually holds in all the future states. Therefore the condition $\neg q$ can only be visited a finite number of times. Dually, a counterexample of $\mathbf{FG}q$ is

an infinite path where $\neg q$ is visited an infinite number of times, i.e. there is a trace satisfying $\mathbf{GF}\neg q$. Since we are working in the finite-state case, such a counterexample must be a lasso-shaped path, i.e. an infinite sequence of states consisting of a finite prefix leading to a cycle that repeats forever. The general model checking problem, denoted by $\mathcal{M} \models \phi$ where ϕ is an LTL formula, can be reduced to the liveness checking problem $\mathcal{M} \times \mathcal{A}_{\neg\phi} \models \mathbf{FG}q$ following the standard automata-theoretic approach [29], where $\neg q$ is the Buchi acceptance condition of $\mathcal{A}_{\neg\phi}$.

3.3 Liveness Checking with rlive

rlive is a recent algorithm for verifying liveness properties in finite-state symbolic transition systems [30]. It can be seen as a variant of k-liveness [9] that explores the state-space in a depth-first search manner. Like other approaches, rlive reduces the liveness checking to a sequence of safety checks. Algorithm 1 describes how rlive is implemented using a generic invariant-checking engine. The key innovation is that rlive builds counterexamples to $\mathbf{FG}q$ incrementally through a recursive, depth-first search process, rather than directly searching for lasso-shaped counterexamples. When looking for counterexamples, rlive first finds a path from the initial states to a $\neg q$-state, i.e. a state that satisfies $\neg q$. This happens in the first iteration of the while-loop at line 4, Algorithm 1, where $\neg C$ and $\neg C'$[1] both evaluate to $\top$. The state s, line 5, is the first $\neg q$-state met. Notice that if such a state is not reachable, then $\mathbf{G}q$ is proved and so is $\mathbf{FG}q$. Then it searches for additional $\neg q$-states from the successors of each discovered $\neg q$-state (line 9). During this process, either of the following outcomes occurs:

1. a previously visited $\neg q$-state is met again, creating a lasso-shaped counterexample that violates the liveness property (lines 11-12), or
2. the search reaches a point where no more $\neg q$-states can be reached. In this case, rlive obtains an inductive invariant, the *shoal*, from the safety checker, which describes the set of states from which $\neg q$ can be visited a finite number of times only (lines 15-16). Clearly, no state in the shoal belongs to a counterexample trace.

Hence, the shoals are used to restrict future searches by blocking parts of the system state space. The algorithms excludes the states in C from the transition system by adding the constraint $\neg C \wedge \neg C'$ to T (lines 4 and 9). Additionally, the states to be searched are no longer simply $\neg q$-states, but states in $T^{-1}(\neg C) \cap \neg q$ (lines 4 and 9), i.e. $\neg q$-states that also have successors outside the shoal C, to exclude $\neg q$-states that are proved not to be part of the counterexample. This procedure continues until either all the reachable $\neg q$-states are eliminated, proving the property (line 18), or a lasso-shaped counterexample is found (lines 11-12).

[1] We remark that for a formula A, the primed notation A' represents the set of states that are immediate successors to states satisfying A. This is semantically equivalent to the LTL next operator $\mathbf{X}$ introduced in Section 3.2

Algorithm 1: rlive algorithm [30].

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

3.4 Theorem Proving in PVS

The Prototype Verification System (PVS) [20] is a specification language integrated with a theorem prover. The PVS theorem prover is interactive, but it also supports strategies development [21] and a batch mode [18], so that proofs can be run automatically. PVS uses a sequent-style [14] proof representation. A PVS sequent is an object of the form $A_1, A_2, A_3, \ldots \vdash B_1, B_2, B_3, \ldots$, where formulae A_i make the antecedent and formulae B_j make the consequent. The sequent above asserts that "if all the A's are true, then at least one of the B's is true". Hence, the sequent means the same as: $(A_1 \wedge A_2 \wedge A_3 \ldots) \rightarrow (B_1 \vee B_2 \vee B_3 \ldots)$.

The prover builds a proof tree that starts with $\vdash A$, where A is the theorem to be established. A proof is accomplished when all the leaves are recognised as true: this occurs if any antecedent is the same as any consequent ($C, \Gamma \vdash C, \Delta$), if any antecedent is false ($\mathit{False}, \Gamma \vdash \Delta$), or if any consequent is true ($\Gamma \vdash \mathit{True}, \Delta$). Other sequents can be recognised as true using more powerful inferences [26].

3.5 A shallow embedding of LTL into PVS

In [28] we present a formalisation of LTL into PVS, following a shallow embedding approach [5, 25].

In the PVS theory `shallow_ltl` we declare the type *trace* as all mappings from natural numbers to states. An *LTL formula* is a function that takes a trace and a natural number, and returns a boolean PVS type (*True* or *False*), which is the truth-value of the formula at point on the trace. A *state* is an object of any type, and it is an explicit parameter of `shallow_ltl`.

```
shallow_ltl[State: TYPE+]: THEORY
BEGIN
Trace: TYPE = ARRAY[nat -> State]
ltlformula: TYPE = [Trace -> [nat -> bool]]
```

Examples of definition of propositional and LTL operators within our theory[2] follow, where P is an LTL formula.

```
NOT(P)(trace: Trace)(t: nat): bool = NOT P(trace)(t);
NEXT(P)(trace: Trace)(t: nat): bool = P(trace)(t+1);
GLOBALLY(P)(trace: Trace)(t: nat): bool = FORALL (t0: nat): t0 >= t IMPLIES P
    (trace)(t0);
```

An LTL formula P is *valid* if it is true at the initial state of any trace. A stronger notion of validity, called *global validity*, is when the formula is true at any state of any trace.

```
|=(trace:Trace, t:nat, P): bool = P(trace)(t)
valid(P): bool = FORALL (trace: Trace): |=(trace, 0, P)
valid_all(P): bool = FORALL (trace: Trace): FORALL (t: nat): |=(trace, t, P)
```

The full theory `shallow_ltl` can be found in a dedicated repository [27].

4 Certifying Liveness Properties Using rlive

Consider the liveness checking problem $\mathcal{M} \models \mathbf{FG}q$, where $\mathcal{M} = \langle X, \mathcal{I}, T\rangle$ and q is a propositional formula over X. With an abuse of notation, we consider T also an LTL formula, identifying x' with $\mathbf{X}(x)$ for every variable $x \in X$. In order to prove $\mathcal{M} \models \mathbf{FG}q$, we provide a proof of $(\mathcal{I} \wedge \mathbf{G}T) \rightarrow \mathbf{FG}q$, following the same approach as in [15].

4.1 A New Temporal Deductive Rule for Liveness

In order to prove $(\mathcal{I} \wedge \mathbf{G}T) \rightarrow \mathbf{FG}q$, we use the following inference rule denoted with RL

$$\frac{P_{\mathtt{i}} \quad P_0 \quad Pk_1 \quad Pp_1 \quad \ldots \quad Pk_n \quad Pp_n}{\mathcal{I} \wedge \mathbf{G}(T) \rightarrow \mathbf{FG}q}\ \mathsf{RL}$$

The premises of the rule RL are:

$P_{\mathtt{i}} := (\mathcal{I} \wedge \mathbf{G}T \wedge \mathbf{G}\neg C) \rightarrow \mathbf{G}q$
$P_0 := \mathbf{G}(C_0 \leftrightarrow \bot)$
$Pk_1 := \mathbf{G}((C_0 \vee C_1) \wedge T \rightarrow \mathbf{X}(C_0 \vee C_1))$
$Pp_1 := \mathbf{G}((C_0 \vee C_1) \wedge T \wedge \neg q \rightarrow \mathbf{X}(C_0))$
$\ldots$
$Pk_n := \mathbf{G}((C_0 \vee \ldots \vee C_n) \wedge T \rightarrow \mathbf{X}(C_0 \vee \ldots \vee C_n))$

[2] PVS allows overloading of built-in symbols. In the definition above the first `NOT` is our defined LTL operator, which creates an LTL formula and whose semantics is defined via the boolean PVS operator `NOT`.

$$Pp_n := \mathbf{G}((C_0 \vee \ldots \vee C_n) \wedge T \wedge \neg q \rightarrow \mathbf{X}(C_0 \vee \ldots \vee C_{n-1}))$$

where $C := C_0 \vee C_1 \vee \ldots \vee C_n$.

Intuitively, $P_{\mathtt{i}}$ means that any trace of $\mathcal{M}$ satisfies that either a state in the shoal will be met eventually, or q is an invariant - being the formula $(\mathcal{I} \wedge \mathbf{G}T \wedge \mathbf{G}\neg C) \rightarrow \mathbf{G}q$ equivalent to $(\mathcal{I} \wedge \mathbf{G}T) \rightarrow (\mathbf{F}C \vee \mathbf{G}q)$. If the latter is the case, then $\mathbf{FG}q$ holds and the liveness property is therefore verified. Thus we need to cover the case where a shoal state is met eventually ($\mathbf{F}C$).

We consider the additional premises of RL, P_0, Pk_1, Pp_1, ..., Pk_n, Pp_n. P_0 simply states that the shoal is initially empty. Each premise Pk_i $(1 \leq i \leq n)$ states that the invariant $C_0 \vee \cdots \vee C_i$ incrementally built is inductive. Each premise Pp_i $(1 \leq i \leq n)$ states that if we are in a state where $C_0 \vee \cdots \vee C_i$ and $\neg q$ both hold, following the transition T, the next state will belong to at least one shoal that was added to C before C_i itself, i.e. to $C_0 \vee \ldots \vee C_{i-1}$. This means that once in the shoal, we do not exit it, and that the search space can be incrementally restricted, as long as we keep visiting a $\neg q$-state. Notice that Pk_1 is equivalent to $\mathbf{G}(C_1 \wedge T \wedge \neg q \rightarrow \bot)$: states in C_1 cannot reach $\neg q$-states at all. C_1 represents the first non-empty set of states added to C by the algorithm.

A formal proof of the fact that rlive contains the information necessary to prove premises $P_{\mathtt{i}}$, and Pk_i, Pp_i for $1 \leq i \leq n$ is given in Section 4.3.

RL as a generalisation of k-liveness rule In the temporal proof for k-liveness from [15] we have formulae $\alpha_0, \ldots, \alpha_{k+1}$, that keep count of the number of times the fairness condition $\neg q$ is reached. Assuming by contradiction that we will keep reaching $\neg q$, eventually α_{k+1} is reached. This final formula expresses a contradiction as $\neg q$ can be visited at most k times by the k-liveness algorithm [9]. Thus, any path starting from $\mathcal{I}$ can visit $\neg q$ finitely many times only (concluding $\mathbf{FG}q$).

RL generalises this k-liveness rule, in the sense that it can be used to build proofs for k-liveness, but the premises are more relaxed to cover more general proofs. In particular, given the α's conditions from k-liveness, RL can be used to perform a k-liveness proof as in [15].

Given $n = k + 1$, we can establish this mapping: each k-liveness condition α_i for $0 \leq i \leq k+1$ maps to C_{k-i+1}, so specifically α_{k+1} maps to C_0, both formulae expressing a contradiction, α_k maps to C_1 and so on through the sequence up α_1 mapping to C_k and α_0 mapping to C_{k+1}. $C = C_0 \vee \ldots \vee C_n = \alpha_{k+1} \vee \ldots \vee \alpha_0$. Given this mapping, we can prove that if the premise $P_{\mathtt{i}}$ of k-liveness holds, which states that $\mathcal{I} \rightarrow \mathbf{F}(\alpha_0)$, than the corresponding $P_{\mathtt{i}}$ of RL holds too, as α_0 implies C using the mapping above. Moreover it is possible to prove by induction that, given this mapping, if each Pk_i and Pp_i premises from k-liveness hold, then so do the corresponding premises for RL.

Thus, RL can be used in alternative to the rule defined in [15] for k-liveness, but the premises are more relaxed to accommodate the proof of rlive. In particular, it provides a more general first premise $P_{\mathtt{i}}$, and weaker premises Pk_i than the corresponding premises of k-liveness. In the k-liveness rule, $P_{\mathtt{i}} := \mathcal{I} \rightarrow \mathbf{F}(\alpha_0)$,

i.e., from the initial state we can reach α_0 and start counting. In RL we need to consider the alternative possibility that the shoal stays empty ($\mathbf{F}C$ is false), thus concluding $\mathbf{G}q$. The Pk_i premises from RL allows to transition from a C_i to any shoal with a lower index, whilst in the k-liveness rule a state from α_i is required to either remain in α_i (when q holds) or transition to the immediate successor condition α_{i+1}.

4.2 Correctness of the rule

Let us denote the set of formulae $\{Pk_1, Pp_1, \ldots, Pk_n, Pp_n\}$ simply with Π. The full formalisation and proof of correctness of RL has been done in PVS in our theory `lemmas_shallow_ltl` [27], in the form of a validity statement: for all formulae $\mathcal{I}$, T, q and C we proved that $valid((P_{\mathtt{i}} \wedge P_0 \wedge \Pi) \rightarrow (\mathcal{I} \wedge \mathbf{G}T \rightarrow \mathbf{FG}q))$. The schema of the proof is as follows, and we refer to [27] for the fine-grained proof.

$$\dfrac{\dfrac{\dfrac{P_{\mathtt{i}}}{(\mathcal{I} \wedge \mathbf{G}T) \rightarrow (\neg\mathbf{G}\neg C \vee \mathbf{G}q)}}{(\mathcal{I} \wedge \mathbf{G}T) \rightarrow (\mathbf{F}C \vee \mathbf{G}q)} \qquad [\mathcal{I} \wedge \mathbf{G}T]}{\mathbf{F}C \vee \mathbf{G}q}$$

If $\mathbf{G}q$ is the case:

$$\dfrac{\dfrac{\mathbf{G}q}{\mathbf{FG}q}}{(\mathcal{I} \wedge \mathbf{G}T) \rightarrow \mathbf{FG}q}$$

Let us now consider the second possibility: $\mathbf{F}C$.

$$\mathsf{RLB}\ \dfrac{\mathbf{F}C \qquad \Pi \qquad [\mathcal{I} \wedge \mathbf{G}T] \qquad [\mathbf{GF}\neg q] \qquad \dfrac{P_0}{\mathbf{G}\neg C_0}}{\dfrac{\mathbf{F}C_0}{\dfrac{\dfrac{\dfrac{\bot}{\neg\mathbf{GF}\neg q}}{\mathbf{FG}q}}{(\mathcal{I} \wedge \mathbf{G}T) \rightarrow \mathbf{FG}q}}}$$

Notice that the main step to prove the correctness of RL is the following deduction rule

$$\mathsf{RLB}\ \dfrac{\mathbf{F}C \qquad \Pi \qquad \mathcal{I} \wedge \mathbf{G}T \qquad \mathbf{GF}\neg q}{\mathbf{F}C_0}$$

This rule states that if we are on a trace where the shoal C is eventually entered, and where $\neg q$ holds infinitely often then, considering the additional premises Π of RL, we are bound to enter the last shoal C_0. As shown in the proof sketch above, the assumption of the fact that $\neg q$ holds infinitely often is used to perform a proof by contradiction, and it is negated when the contradiction is reached. Also RLB has been formalised and proved in PVS within our theory `lemmas_shallow_ltl` [27]. The proof RLB uses the KLB rule from [15], which is the main step for the deduction of their temporal rule for k-liveness. This is because, as explained earlier, RL is a generalisation of this rule.

4.3 Completeness for rlive

In this section, we show that the rule is complete to provide a proof for the property proved by rlive, in the sense that rlive can be easily extended to generate the premises of the rule.

The C_i are the inductive invariants that are generated by performing a series of invariant checks on variations of the original transition system. Thus, initially $C := C_0$ is empty ($C_0 = \bot$, from line 2 of Algorithm 1). New shoals are added in disjunction and at the i-th iteration $C = C_{\leq i} := C_0 \vee \ldots \vee C_i$ (lines 15-16). At each iteration (lines 8-17), rlive proves that the new C_{i+1} shoal is an inductive invariant for the modified transition $T \wedge \neg C_{\leq i} \wedge \neg C'_{\leq i}$ and that it implies the invariant $T \wedge \neg C'_{\leq i} \rightarrow q$ (line 9). We can prove that this is sufficient to prove the premises Pk_n and Pp_n from section 4.2.

Theorem 1. *Assume that for all i, $0 \leq i < n$, the following holds:*

$$\models (C_{i+1} \wedge T \wedge \neg C_{\leq i} \wedge \neg C'_{\leq i}) \rightarrow C'_{i+1} \tag{1}$$

$$C_{i+1} \wedge \neg q \wedge T \wedge \neg C'_{\leq i} \models \bot \tag{2}$$

Then for all i, $0 \leq i < n$, the following implications are valid:

$$((C_{\leq i} \vee C_{i+1}) \wedge T) \rightarrow (C'_{\leq i} \vee C'_{i+1}) \tag{3}$$

$$((C_{\leq i} \vee C_{i+1}) \wedge T \wedge \neg q) \rightarrow C'_{\leq i} \tag{4}$$

Proof. We prove (3) by induction on i. Since $C_{\leq i} = \bot$ in case $i = 0$, the base case of (3), what we want to prove says that $(C_1 \wedge T) \rightarrow (C'_1)$, which is exactly the assumption (1) with $i = 0$.

Let us consider the step case of the induction. By inductive hypothesis we know that $(C_{\leq i} \wedge T) \rightarrow (C'_{\leq i})$. By (1), we have that $\neg C_{\leq i} \wedge C_{i+1} \wedge T \rightarrow C'_{\leq i+1}$. From these two, we can deduce that the same holds for $((C_{\leq i} \vee C_{i+1}) \wedge T) \rightarrow (C'_{\leq i} \vee C'_{i+1})$, that is 3.

We now prove (4). From (3), we have that $((C_{\leq i}) \wedge T) \rightarrow (C'_{\leq i})$. From (2), we deduce that $(C_{i+1} \wedge T \wedge \neg q) \rightarrow C'_{\leq i}$. From these two, we can deduce that $((C_{\leq i} \vee C_{i+1}) \wedge T \wedge \neg q) \rightarrow C'_{\leq i}$, that is (4). □

Similarly, we can prove that the information provided by the algorithm rlive is sufficient to prove the initial premise $P_{\mathtt{i}}$. The idea is that C represents the final shoals returned by the rlive algorithm and the algorithm proves that $P_{\mathtt{i}}$ holds. The last invariant check at line 4 of Algorithm 1 returns an invariant, let us call it ψ, that is inductive for the modified transition $T \wedge \neg C \wedge \neg C'$, and such is that it implies the invariant $T \wedge \neg C' \rightarrow q$.

Theorem 2. *Assume that the following holds:*

$$\models \psi \rightarrow (T \wedge \neg C' \rightarrow q) \tag{5}$$

$$\models \mathcal{I} \wedge \mathbf{G}(T \wedge \neg C) \rightarrow \mathbf{G}\psi \tag{6}$$

Then the following implication is valid:

$$\mathcal{I} \wedge \mathbf{G}(T \wedge \neg C) \rightarrow \mathbf{G}q \tag{7}$$

Proof. From 5 and 6 it immediately follows that $\mathcal{I} \wedge \mathbf{G}(T \wedge \neg C) \rightarrow \mathbf{G}(T \wedge \neg C' \rightarrow q)$. This is equivalent to $\mathcal{I} \wedge \mathbf{G}(T \wedge \neg C) \rightarrow \mathbf{G}(q \vee \neg T \vee C')$. From this we can easily prove that $\mathcal{I} \wedge \mathbf{G}(T \wedge \neg C) \rightarrow \mathbf{G}q$ which is 7. This final step has been verified in PVS, since it is an essential proven lemma of our proof strategy presented in Section 4.5. Having proven it definitively, we can reuse it throughout our approach. For the detailed proof, we refer to [27], where the proven lemma appears under the name `stronger_ind_proof_2`. □

These proofs give an alternative demonstration of the correctness of the rlive algorithm, when the property is proved correct, using the rule RL. The proofs are given here for completeness of the presentation. However, to avoid trusting the implementation of the rlive algorithm and in the spirit of certifying model checking, we set a certification process in which we use a theorem prover to check the certificates generated by the model checker, as detailed in the following section.

4.4 Certification process

Our certification process goes through three main stages:

1. The model checking stage, where we run the model checker and we dump the inductive invariants C making the shoals and the final invariant ψ as described in Section 4.3. These are the parameters required by the proof strategy described in the next section.
2. The theory generation phase, where a PVS theory is generated with the relevant specification of the model $\mathcal{M}$, the property to be proved, the parameters, the claim of the main theorem and the PVS proof-script with the strategy to be run to prove the main theorem.
3. The PVS proof which generates the proof certificate. Each proof uses the PVS strategy presented in the next sections, and follows a consistent pattern for all liveness checking problems $\mathcal{M} \models \mathbf{FG}q$. The proof consists of two key components: a *structural part* and a *proof obligations discharging part*. The structural part involves applying LTL definitions and the temporal deductive rules which we have proved *once and for all* as PVS lemmas, such as the lemmas for rlive RL. This part remains identical regardless of the specific model and property being certified. The proof obligations discharging part focuses on proving the propositional implications at the leaves of the proof tree and it is accomplished using the PVS built-in SAT solver (PVS uses the SMT solver Yices [11]). The critical aspect of the proof lies in this discharge of the proof obligations by Yices, as these proof steps confirm that the model $\mathcal{M}$ satisfies the necessary premises for applying RL, thereby validating the conclusion of $\mathcal{M} \models \mathbf{FG}q$. We remark that, alternatively to a SAT solver, purely syntactic and resolution based methods can be used in this stage of the proof strategy to discharge the propositional implication leaves. In PVS such methods are prop or bdd-simp [26]. However, for large formulae, they do not scale as well as a SAT solver. A full example of a proof using RL, on a concrete transition system and liveness property, can be found at [27] (`rlive_example_proof_output`).

4.5 A Proof Strategy for Liveness Checking

Given a model $\mathcal{M} = \langle X, \mathcal{I}, T \rangle$, a liveness property q and a formula C representing the shoal, which is provided by the model checking stage, a proof strategy for certifying liveness results utilises the temporal deductive rule RL (as presented above) as a proven lemma, and then proceeds to discharge each of its premises $P_{\mathtt{i}}, P_0$ and Π.

The strategy takes the sequence of formulae $C_0, \ldots, C_n$, whose disjunction makes the shoal, as explicit parameters. The model checker provides an additional parameter formula, let us call it ψ. As we will see later, this is the inductive invariant that will be used to accomplish the proof of the premise $P_{\mathtt{i}}$.

The proof tree starts with the goal: $\vdash \mathit{valid}(\mathcal{I} \wedge \mathbf{G}T \rightarrow \mathbf{FG}q)$. The strategy splits in two branches: **Branch 1**, where the premise $P_{\mathtt{i}}$ is added to the set of assumptions, and **Branch 2** where $P_{\mathtt{i}}$ is added to the set of conclusions, to be proved. In practice we make use of the PVS rule `Case` which allows us to assume a formula and subsequently prove this formula to be true [26].

Branch 1 in turns splits into two branches: **Branch 1.1** where the remaining premises P_0 and Π's are added to the set of assumptions, and **Branch 1.2** where P_0 and Π's are added to the set of conclusions, to be proved.

On **Branch 1.1** RL is added to the set of assumptions as a proven lemma, with appropriate substitution for the formulae representing the shoal C, $\mathcal{I}$, T and the liveness property q within the RL premises $P_{\mathtt{i}}$, P_0 and Π. The sequent at the leaf of **Branch 1.1** has this form:

$$
\begin{array}{r|l}
-1 & \mathit{valid}(P_{\mathtt{i}} \wedge P_0 \wedge \Pi \rightarrow (\mathcal{I} \wedge \mathbf{G}T \rightarrow \mathbf{FG}q)) \\
-2 & \mathit{valid}(P_{\mathtt{i}}) \\
-3 & \mathit{valid}(P_0 \wedge \Pi) \\
\hline
1 & \mathit{valid}(\mathcal{I} \wedge \mathbf{G}T \rightarrow \mathbf{FG}q)
\end{array}
$$

which is clearly provable after expanding the definition of "*valid*" and some simple symbolic manipulation. This concludes the proof of **Branch 1.1**.

The strategy turns then to **Branch 1.2**, where the premises P_0, Pk_1, Pp_1, $\ldots$, Pk_n, Pp_n have to be discharged. After expanding the definition of the *Globally* operator $\mathbf{G}$, which is the main operator of each of these premises, they can all be discharged by rewriting $\mathcal{I}$, T, q and each C_i for $0 \leq i \leq n$ with the appropriate formulae from the theory at issue, and by using the PVS SAT solver Yices to prove the propositional implications at the resulting leaves. This completes the proof of **Branch 1.2**.

The strategy turns to **Branch 2**, where premise $P_{\mathtt{i}}$ has to be discharged. We remind the reader that $P_{\mathtt{i}} := \mathcal{I} \wedge \mathbf{G}T \wedge \mathbf{G}\neg C \rightarrow \mathbf{G}q$. This is a invariant claim equivalent to $\mathcal{I} \wedge \mathbf{G}(T \wedge \neg C) \rightarrow \mathbf{G}q$. $P_{\mathtt{i}}$ expresses that the formula q is an invariant for the model $\hat{\mathcal{M}} = \langle X, \mathcal{I}, \hat{T} \rangle$ with $\hat{T} := T \wedge \neg C \wedge \neg C'$. Since formula q is not necessarily inductive w.r.t. $\hat{\mathcal{M}}$, we need an invariant formula ψ, that is inductive and that will imply the proof of the invariant claim in $P_{\mathtt{i}}$. The

model checker is able to produce such an inductive invariant as the result of the invariant check contained within the rlive algorithm, as described in Section 4.3. (line 4 of Algorithm 1). It then passes this as a parameter for the proof strategy, together with the shoal formula C. In order to prove $P_{\mathtt{i}}$, the proof strategy applies a subroutine for proving invariants using inductive invariants. This subroutine strategy takes the formula ψ as parameter and carries out a proof of $P_{\mathtt{i}}$. This strategy is an adaptation of the strategy for proving invariant presented in [28]. This completes the proof of **Branch 2**.

Our PVS implementation of this strategy can be found at [27], together with a full example of a proof using this strategy.

5 Experimental Evaluation

In this section, we present the experimental evaluation of our proposed methodology, assessing the effectiveness, efficiency, and robustness of our approach through a series of comprehensive tests run on publicly available benchmarks sets. We describe the experimental setup, including the hardware and software configurations, the datasets used, and the specific metrics considered for the evaluation. We also provide a detailed analysis of the results obtained.

Setup We have implemented our proof generation and certification procedure on top the model checking tool nuXmv [7]. The tool takes as input a model in Aiger [2] format, and produces the inductive invariants making the shoal and the inductive invariant ψ described in Section 4.5, as Aiger combinational circuits expressed over the state variables of the model.

We then apply a simple Python script to translate the input model and the generated invariants into a PVS theory. We make the script available (together with the rest of the our toolchain) at [27]. The Python script is relatively simple, and it can be verified through standard software verification methodologies. This was adapted from our work from [28] to include generation of theories for liveness checking. The overhead of this translation stage (namely, stage 2 of the certification process described in Section 4.4) is negligible, and therefore not included in the results evaluation where we only compare the model checking stage and the proof generation stage.

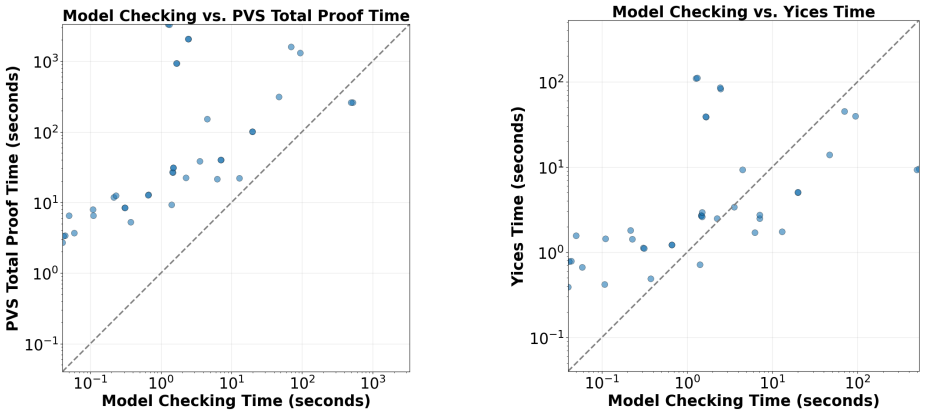
Benchmark Set For our evaluation, we have collected a total of 53 distinct problem instances from different families, stemming from previous Hardware Model Checking Competitions (HWMCC) [3]. These 53 instances are the safe instances that could be successfully proved by nuXmv in the time limit of 1200 seconds. All benchmarks, certificates, and run logs have been made available for reference [27].

Results The nuXmv runs were carried out on a computer cluster, on a queue consisting of 4 nodes with identical hardware specifications. Each node is equipped

with an Intel Xeon CPU 6226R processor operating at 2.9 GHz, with 32 CPU cores and 16 GB of memory. These jobs were managed by SLURM with a memory limit of 4 GB and a wall-time limit of 1200 seconds per experimental run.

PVS experiments were conducted using PVS8.0[3] on the set of benchmarks described above, on the same cluster and in the same setup mentioned above, with a memory limit of 8 GB and a wall-time limit of 3600 seconds per run.

The results demonstrate the effectiveness of the prototype implementation, integrating our rlive proof strategy described in Section 4.5 within the existing certifying model checking framework based on PVS [28]: of the 53 total test cases, 41 in total were successfully proved by PVS within the allotted time and memory constraints, whilst remaining 12 cases exceeded the available memory resources.



(a) (b)

Fig. 1: Comparison between model checking verification and PVS certification times, considering overall certification times (a) and just SAT solver times (b).

Figure 1 presents preliminary results comparing model checking instances verification times, i.e. the time taken by the model checker to verify that an instance model satisfies a liveness property using the rlive algorithm (this is stage 1 of the certification process described in Section 4.4), against the certification times of PVS, i.e. the time taken by PVS to prove the corresponding theorem (this is stage 3 of the certification process described in Section 4.4). Figure 1a accounts for all the PVS certification contributions whereas Figure 1b focuses on just Yices SAT solver times. Figure 1a shows that almost all data points lie significantly above the diagonal dashed line indicating that PVS total proof

[3] We modified the PVS Makefile configuration to increase resource allocations. The `SBCL_SPACE_SIZE` parameter was increased from 6 GB to 30 GB, and `SBCL_STACK_SIZE` was increased from 8 MB to 32 MB.

time is consistently higher than the model checking verification time for the same problems. Figure 1b shows that Yices times are generally closer to the diagonal line, suggesting that the proof-obligation discharging part of the proof is more efficient than the rest of the steps performed in the PVS proofs.

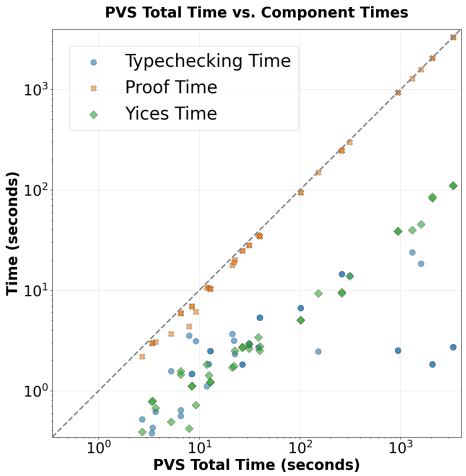


Fig. 2: Times breakdown for PVS-based runs.

To better understand the reasons behind this gap, we further analysed the breakdown of execution times for different operations within PVS. Figure 2 illustrates the total certification time divided into three components: *type-checking time*, the time spent verifying semantic constraints, determining expression types, and resolving names, *proof-checking time*, the time spent executing the actual steps of the proof strategy and *SAT-solving time*, the time of the proof-checking time spent on calls to the SAT solver Yices. Our analysis reveals that PVS performance bottlenecks occur primarily within proof steps other than the Yices solver component. When processing theories with numerous shoals formulae, PVS spends disproportionate time on fundamental logical operations, such as logical operators' expansion. For n shoals, expanding basic LTL operators like 'AND' requires approximately n separate expansion operations, each triggering cascading expansions of nested 'OR' operations. These operations, as well as operations of installing and rewriting definitions, involve substantial hidden costs. PVS performs additional typechecking and book-keeping during these operations. This explains the significant gap between Yices processing time and proof time observed in our performance analysis.

While our analysis indicate scaling limitations in the current prototype implementation, it is important to emphasise that the primary contribution of this work is the theoretical foundation and correctness of our proof strategy for rlive. The prototype serves its intended purpose: demonstrating that the strategy is

sound and functional by successfully certifying the tested benchmarks. The PVS based framework could be implemented using alternative certification tools or optimised versions of PVS tailored to this problem domain. As proposed in [28], future work will address these scaling concerns.

6 Conclusions and Future Work

In this paper we have considered rlive [30], a new SAT-based model-checking algorithm for the verification of liveness properties of finite-state symbolic transition systems. We have shown that, despite the complexity of the algorithm, the shoals provided by the model checker are sufficient to generate proof certificates in a deductive system for temporal properties, and that even though rlive and k-liveness seem two very different algorithms, in terms of temporal rules the first can be seen a generalisation of the second, as it is presented in [15]. We have formalised the rule for rlive, proved its correctness and completeness, and developed a proof strategy that uses this rule to certify liveness checking results. We have implemented our strategy as a prototype in an existing certifying model checking framework [28] based on PVS, and tested the implementation on a set of benchmarks from the hardware model checking competition.

We see several directions for future work, such as extending the certifying model checking approach to other liveness checking algorithms such as liveness-to-safety [1] and FAIR [6]. It would also be interesting to investigate the possibility of a proof strategy that encompasses all these liveness checking algorithms. We would also like to generalise the current proof strategy for rlive to account for multiple fairness constraints, as it is done for k-liveness in [15], since our approach is currently limited to a single fairness condition (namely $\neg q$). Since proofs are easily composed, it is a very feasible next step to consider the generation of proofs that combine multiple transformation techniques with liveness proofs, as it is done in [15] for k-liveness. Some of these transformations, such as temporal decomposition and phase abstraction, are already certifiable using the theorem prover based approach from [28]. We would also like to consider generalisations to the infinite-state transition system and SMT (Satisfiability Modulo Theories) based model checking, since rlive has recently been generalised to handle this type of transition systems [8].

Acknowledgments. The authors acknowledge the support of the PNRR project FAIR - Future AI Research (PE00000013), under the NRRP MUR program funded by the Next Generation EU and in particular the activity of technology transfer "nuXmv - Towards Market Readiness".

References

1. Armin Biere, Cyrille Artho, and Viktor Schuppan. Liveness checking as safety checking. *Electronic Notes in Theoretical Computer Science*, 66(2):160–177, 2002.

2. Armin Biere, Keijo Heljanko, and Siert Wieringa. AIGER 1.9 and beyond. Technical Report 11/2, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, 2011.
3. Armin Biere and Toni Jussila. The Model Checking Competition Web Page, http://fmv.jku.at/hwmcc.
4. Armin Biere, Emily Yu, and Nils Froleyks. Stratified certification for k-induction. In *Proceedings of the 22nd Conference on Formal Methods in Computer-Aided Design–FMCAD 2022*, volume 3, page 59. TU Wien Academic Press, 2022.
5. Richard J Boulton, Andrew D Gordon, Michael JC Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in hol. In *TPCD*, volume 10, pages 129–156, 1992.
6. 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.
7. Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuxmv symbolic model checker. 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 334–342. Springer, 2014.
8. Alessandro Cimatti, Alberto Griggio, Christopher Johannsen, Kristin Y. Rozier, and Stefano Tonetta. Infinite state liveness checking with rlive. In *Proceedings of the 37th International Conference on Computer Aided Verification (CAV 2025)*, 2025. To be published.
9. 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.
10. Christian Dax, Martin Hofmann, and Martin Lange. A proof system for the linear time μ-calculus. In *FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science: 26th International Conference, Kolkata, India, December 13-15, 2006. Proceedings 26*, pages 273–284. Springer, 2006.
11. Bruno Dutertre and Leonardo De Moura. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2(2):1–2, 2006.
12. Nils Froleyks, Emily Yu, Armin Biere, and Keijo Heljanko. Certifying phase abstraction. In *International Joint Conference on Automated Reasoning*, pages 284–303. Springer, 2024.
13. Dov Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the temporal analysis of fairness. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 163–173, 1980.
14. Gerhard Gentzen. Untersuchungen über das logische schließen. i. *Mathematische zeitschrift*, 35, 1935.
15. Alberto Griggio, Marco Roveri, and Stefano Tonetta. Certifying proofs for sat-based model checking. *Formal Methods in System Design*, 57(2):178–210, 2021.
16. Tuomas Kuismin and Keijo Heljanko. Increasing confidence in liveness model checking results with proofs. In *Haifa Verification Conference*, pages 32–43. Springer, 2013.
17. Orna Kupferman and Moshe Y Vardi. From complementation to certification. *Theoretical computer science*, 345(1):83–100, 2005.
18. César A Munoz. Batch proving and proof scripting in pvs. Technical report, National Institute of Aerospace, 2007.

19. Kedar S Namjoshi. Certifying model checkers. In *Computer Aided Verification: 13th International Conference, CAV 2001 Paris, France, July 18–22, 2001 Proceedings 13*, pages 2–13. Springer, 2001.
20. Sam Owre, John M. Rushby, and Natarajan Shankar. Pvs: A prototype verification system. In *International Conference on Automated Deduction*, pages 748–752. Springer, 1992.
21. Sam Owre and Natarajan Shankar. Writing pvs proof strategies. In *Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003), number CP-2003-212448 in NASA Conference Publication*, pages 1–15, 2003.
22. Doron Peled, Amir Pnueli, and Lenore Zuck. From falsification to verification. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 292–304. Springer, 2001.
23. Doron Peled and Lenore Zuck. From model checking to a temporal proof. In *International SPIN workshop on model checking of software*, pages 1–14. Springer, 2001.
24. Amir Pnueli. The temporal logic of programs. In *18th annual symposium on foundations of computer science (sfcs 1977)*, pages 46–57. ieee, 1977.
25. John Rushby. Pvs embeddings of propositional and quantified modal logic. *arXiv preprint arXiv:2205.06391*, 2022.
26. Natarajan Shankar, Sam Owre, John M Rushby, and Dave WJ Stringer-Calvert. Pvs prover guide. *Computer Science Laboratory, SRI International, Menlo Park, CA*, 1:11–12, 2001.
27. Giulia Sindoni, Alberto Griggio, and Stefano Tonetta. Certifying-rlive-25. `https://gitlab.fbk.eu/gsindoni/Certifying_Rlive_25`. Accessed: 2025-05-15.
28. Giulia Sindoni, Paolo Pasini, Gianpiero Cabodi, Paolo E. Camurati, Alberto Griggio, Marco Palena, Marco Roveri, and Stefano Tonetta. A theorem prover based approach for sat-based model checking certification. In *Automated Deduction-CADE-35: 30th International Conference on Automated Deduction, Stuttgart, Germany, 2025, Proceedings 30*, 2025. To be published.
29. Moshe Y Vardi. An automata-theoretic approach to linear temporal logic. *Logics for concurrency: structure versus automata*, pages 238–266, 2005.
30. 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.
31. Emily Yu, Armin Biere, and Keijo Heljanko. Progress in certifying hardware model checking results. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II 33*, pages 363–386. Springer, 2021.