A Theorem Prover Based Approach for SAT-Based Model Checking Certification

Giulia Sindoni✉[1][0000−0003−4003−2317], Paolo Pasini✉[2][0000−0001−6233−0994], Gianpiero Cabodi[3][0000−0001−5839−8697], Paolo E. Camurati[3][0000−0002−2476−2160], Alberto Griggio✉[1][0000−0002−3311−0893], Marco Palena[4][0000−0003−0605−9014], Marco Roveri[5][0000−0001−9483−3940], and Stefano Tonetta✉[1][0000−0001−9091−7899]

[1] *FBK* - Fondazione Bruno Kessler, Trento, IT, `{gsindoni,griggio,tonettas}@fbk.eu`
[2] *DET* - Dept. of Electronics and Telecommunications, *Politecnico di Torino*, IT `paolo.pasini@polito.it`
[3] *DAUIN* - Dept. of Control and Computer Engineering, *Politecnico di Torino*, IT `{gianpiero.cabodi,paolo.camurati}@polito.it`
[4] *CNIT* - National Inter-University Consortium for Telecommunications, Torino, IT `marco.palena@polito.it`
[5] Dept. of Information Engineering and Computer Science, *University of Trento*, IT `marco.roveri@unitn.it`

Abstract. In the field of formal verification, certifying proofs serve as compelling evidence to demonstrate the correctness of a model within a deductive system. These proofs can be automatically generated as a by-product of the verification process and are key artifacts for high-assurance systems. Their significance lies in their ability to be independently verified by proof checkers, which provides a more convenient approach than certifying the tools that generate them. Modern model checking algorithms adopt deductive methods and usually generate proofs in terms of inductive invariants, assuming that these apply to the original system under verification. Model checkers, though, often make use of a range of complex pre-processing simplifications and transformations to ease the verification process, which add another layer of complexity to the generation of proofs. In this paper, we present a novel approach for certifying model checking results exploiting a theorem prover and a theory of temporal deductive rules that can support various kinds of transformations and simplification of the original circuit. We implemented and experimentally evaluated our contribution on invariants generated using two state-of-the-art model checkers, nuXmv and PdTRAV, and by defining a set of rules within a theorem prover, to validate each certificate.

Keywords: Model Checking, Certificate, Safety, Invariant, Proof Checking

1 Introduction

Adopting formal methods as a tool for certifying high-assurance and safety-critical systems calls for verification tools to being capable of providing a high

level of confidence in their outcomes. Verification tools such as model checkers, are inherently complex: they rely on a variety of heuristics, transformation and simplification steps and they often exploit a variety of different techniques and algorithms in order to successfully tackle verification instances.

A model checker generally offers a straightforward “yes” or “no” answer when addressing a verification problem. However, the growing complexity of model checkers themselves has made it increasingly important to obtain certificates from the model checking process. These certificates, often in the form of deductive proofs, serve to build trust in the verification results by providing additional evidence of correctness.

The idea is for model checkers to behave similarly to SAT solvers within the SAT community. SAT solvers provide certificates that serve as proofs of unsatisfiability when a formula cannot be satisfied. These certificates enhance the trustworthiness of the solvers’ outcomes, ensuring that the results can be independently verified by third parties.

For model checking, certification would involve generating deductive proofs as a by-product of the verification process [27], to be then independently verified by proof checkers, which are generally simpler to certify compared to the original proof-generating tools. The independent verification process further boosts the reliability of the results, since it can be conducted at any time outside of the model checker context.

To be effective on real-world instances, though, most model checkers adopt a range of pre-processing simplifications. For example, the model checker may compute some equivalences among variables of the original system and exploit them to reduce the model. As a result, the invariants generated by these model checkers are inductive relative to other invariants, meaning that they hold true based on the assumptions made about these other invariants. Other simplifications decompose temporally the problem: temporal decomposition first proves the invariant on the first k steps and then finds an inductive invariant that holds only after k steps [14]; phase abstraction [6] instead proves a property on subsequences defined by periodic signals. Henceforth, the generation of proofs becomes much more complex in these cases, as it requires temporal reasoning beyond the simple induction principle.

In this paper, we strive to reduce the burden of model checker developers for generating certifying proofs, by providing a proof system based on an existing theorem prover, namely PVS [29], that would build and prove theorems of the system under verification based on the information provided by a model checker. The proof system is based on a theory of temporal deductive rules for Linear-time Temporal Logic (LTL), accounting for the temporal reasoning behind simplifications. We develop strategies in PVS to prove safety properties using equivalences, temporal decomposition, and phase abstraction. This allows model checkers to generate certificates with minimal overhead – essentially outputting an inductive invariant for the simplified system along with a small amount of additional information (e.g., the bound used for temporal decomposition or the period for phase abstraction) – while delegating the proof process to the theorem prover.

We implemented and experimentally evaluated our contribution on invariants generated using two state-of-the-art model checkers, nuXmv and PdTRAV, and by defining a set of rules within PVS, to validate each certificate. The combination of two model checkers for proof generation, coupled with PVS for certification, aims at supporting the rigour and adaptability of the proposed approach.

Outline The rest of the paper is organized as follows: §2 provides an overview of related work. §3 introduces notation and background notions. §4 describes our proposed approach at theorem prover based certification. §5 reports an evaluation of the proposed approach on industrial-level benchmarks, and §6 concludes the paper.

2 Related Work

This work leverages existing literature and further delves into the issue of certifying model checking results. This work extends on the concepts first presented in [23, 24], where it was shown how to exploit the k-liveness algorithm to extend proof generation capabilities for invariant checking to cover full linear-time temporal logic (LTL) properties, with little overhead for the model checker. However, no theorem prover was used to check the proof generated by the model checker.

Among the most related works, [37] introduces a formal framework designed to certify the results of k-induction-based model checking. At the core of such a framework there is the concept of a k-witness circuit, acting as a simulation of the original circuit and including an inductive invariant that serves as a proof certificate. Such an approach is further expanded in [5], where the authors show how to simplify the approach by assuming reset functions are stratified, enabling the approach to be extended to word-level reasoning. In [38] the authors present an approach to certify temporal decomposition, and in [20], the authors present an approach to certify an extended form of phase abstraction using a generic certificate format, still relying on the construction of a witness circuit, which includes an inductive invariant property, that serves as certificate.

The main difference between our approach and [37] is that in the latter every model checker must be extended to generate a witness circuit. In our approach the model checker does not need to perform extra processing, while the proof system builds the proof for every model checker using the supported transformation. The transformation techniques have been proved once and for all as PVS lemmas. Those proofs can be checked, and they are purely symbolic manipulations done in a sequent calculus style. The proved lemmas are then used in the strategies to provide a certificate of the correctness of the fact that the property is satisfied in the corresponding model.

Moreover, [37] is explicitly limited to invariant properties only, whereas our approach can be extended to certifying liveness and general LTL properties.

Therefore, we reduce the burden of model checker developers to produce the certificates themselves. However, checking the witness circuit with a specialised

model checker is more efficient compared to the certification based on an off-the-shelf theorem prover.

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 variables. Uppercase Latin letters F, $\mathcal{I}$, T as well as lowercase Greek letters ϕ, ψ, χ, are used to denote formulae, while uppercase Greek letters Γ and Δ represent sets of formulae. We write $F(X)$ to indicate that all variables appearing in F belong to X , which is referred to as the *support* of F. Likewise, given two disjoint sets of variables X and X', the notation $F(X, X')$ signifies that the support of F is $X \cup X'$.

3.1 Transition Systems

We take into account systems modelled 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 evaluates 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 $i \geq 0$ we have that $s_i, s_{i+1} \models T$. Given a path $\boldsymbol{\pi} := s_0, s_1, \ldots$ we denote with $\boldsymbol{\pi}[i]$ the state s_i.

3.2 Linear Temporal Logic

Linear Temporal Logic (LTL) was introduced by Pnueli [31] 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* (negation $\neg$, conjunction $\wedge$ and disjunction $\vee$) and some *temporal operators* $\mathbf{X}$ (“next time”), $\mathbf{F}$ (“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.

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

- $\boldsymbol{\pi}, i \models \top$ and $\boldsymbol{\pi}, i \not\models \bot$.
- For each $p \in X$, $\boldsymbol{\pi}, i \models p$ iff $\boldsymbol{\pi}[i] \models p$.

- $\boldsymbol{\pi}, i \models \neg\psi$ iff $\boldsymbol{\pi}, i \not\models \psi$.
- $\boldsymbol{\pi}, i \models \psi_1 \wedge \psi_2$ iff $\boldsymbol{\pi}, i \models \psi_1$ and $\boldsymbol{\pi}, i \models \psi_2$.
- $\boldsymbol{\pi}, i \models \psi_1 \vee \psi_2$ iff $\boldsymbol{\pi}, i \models \psi_1$ or $\boldsymbol{\pi} \models \psi_2$.
- $\boldsymbol{\pi}, i \models \mathbf{X}\psi$ iff $\boldsymbol{\pi}, i+1 \models \psi$.
- $\boldsymbol{\pi}, i \models \mathbf{F}\psi$ iff $\boldsymbol{\pi}, j \models \psi$ for some $j \geq i$.
- $\boldsymbol{\pi}, i \models \mathbf{G}\psi$ iff $\boldsymbol{\pi}, j \models \psi$ for every $j \geq i$.
- $\boldsymbol{\pi}, i \models \psi_1 \mathbf{U} \psi_2$ iff $\boldsymbol{\pi}, j \models \psi_2$ for some $j \geq i$ and $\boldsymbol{\pi}, k \models \psi_1$ for every $i \leq k < j$.

Other common Boolean operators have semantics that can be easily inferred from the semantics of $\neg$, $\wedge$ and $\vee$.

Finally, $\boldsymbol{\pi} \models \psi$ iff $\boldsymbol{\pi}, 0 \models \psi$. Given a propositional formula ϕ over X, we call the invariant checking problem, denoted by $\mathcal{M} \models \mathbf{G}(\phi)$, the problem to check $\boldsymbol{\pi} \models \mathbf{G}(\phi)$ for all paths $\boldsymbol{\pi}$ of $\mathcal{M}$.

3.3 Model Transformations

Given a transition system $\mathcal{M}$, several pre-processing and transformation techniques may be applied before proceeding with the actual verification task. When taking into account concrete model instances, representing digital circuits, it is usual to extend the concepts previously introduced to such a context. Hardware combinational circuits, are often represented as directed acyclic graphs (DAGs), where every gate of the circuit is associated with a given Boolean function and state variables for the systems are materialized as memory elements.

Equivalence and Constant Propagation Several pre-processing techniques rely on formulae simplifications through the identification of equivalence classes among variables. Exploiting equivalences and constants propagation one can derive a new model whose behaviour mirrors the original one. One such a technique is ternary simulation [34], that performs a symbolic simulation of the circuit using three-valued logic.

Temporal Decomposition Temporal decomposition [14] is a technique designed to simplify a given input system by identifying and eliminating unnecessary logic associated with initialization or reset sequences within the circuit. This logic typically pertains to operations executed exclusively during the initial phases of the system's operation, and thus could be ignored when evaluating the actual behaviour of the system. The core concept revolves around estimating, or guessing, the duration of the reset sequence. Once such a length k is determined, the technique allows for the modification of the system's initial states, by replacing them with the set of states reachable within k steps, and obtaining a new model for the system. Additional simplifications might be applied alongside temporal decomposition.

Even if no additional simplifications are performed, if the property ϕ under verification is an invariant, in case of temporal decomposition, it is necessary to check whether the property holds in the first k steps as well.

Phase Abstraction Phase abstraction [1, 6] is a simplification technique that operates by detecting and removing periodic signals that exhibit clock-like behaviours and partitioning the system according to the identified phases.

Using phase abstraction, it is possible to remodel the system under verification in order to bypass difficulties that arise from managing multi-phase systems. Considering two-phase designs, the technique operates by replacing the system's transition relation with a composition of two instances of the original transitions. Additionally, it expands the property under verification to include all states reaching a safe state (with respect to the original property) in a single step.

3.4 Theorem Proving in PVS

The Prototype Verification System (PVS) [29] is a specification language integrated with a theorem prover. The PVS theorem prover is interactive, but it also supports strategy development [30] and a batch mode [26], so that proofs can be run automatically. PVS uses a sequent-style [22] 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 [35].

4 Certifying Proofs for SAT-Based Model Checking

In this section we present our approach at theorem prover based certification. Section 4.1 gives a summary of the certification process. Section 4.2 shows how to formalise the LTL logic in PVS. Section 4.3 presents a basic PVS strategy for proving invariants. Section 4.4 formalises phase abstraction as a temporal deductive rule and introduces a PVS strategy for phase abstraction. Section 4.5 presents a PVS strategy for temporal decomposition.

4.1 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 invariants and any other parameter which will be required by the relevant PVS strategy. These parameters are the inductive invariant in the case of the basic strategy with no transformation (see Section 4.3), the inductive invariant $\hat{\chi}$ of the phase-abstracted model (see Section 4.4) in the case where

the model has gone through phase abstraction, and the inductive invariant ι and the natural number k (see Section 4.5) in the case where the model has gone through temporal decomposition.
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 one of the PVS strategies presented in the next sections, and follows a consistent pattern for all invariant checking problems $\mathcal{M} \models \mathbf{G}(\phi)$. 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 phase abstraction or temporal decomposition). 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 [19]).
   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 the applied transformations lemmas, thereby validating the conclusion of $\mathcal{M} \models \mathbf{G}(\phi)$.

We remark that, alternatively to a SAT solver like Yices, purely syntactic and resolution based methods can be used at stage 3 of the process, in order to discharge the propositional implication leaves. In PVS such methods are `prop` or `bddsimp` [35], which we originally used in our strategies, and they successfully generated proof trees with trivially true sequent leaves of the form $A, \Delta \vdash A, \Gamma$. However, these purely syntactical methods exhibited the expected scalability limitations, thus the need for a SAT solver. We would also like to underline that there are SAT solvers capable of producing proofs, which could in turn be independently verified [18, 21, 25]. These could replace Yices, providing an additional layer of certification. While such proof producing SAT solvers are not currently available into the PVS framework, their development and incorporation represents a promising avenue for future work.

Our aim is to reduce a complex problem to a simpler one. We transform the problem of proving invariants through model checking algorithms and transformations techniques, into the verification of propositional formulae that can be discharged by a SAT solver. This represents a significant reduction in complexity, as the propositional formula handled by the SAT solver is a substantially simpler problem than the original model checking problem.

4.2 A Shallow Embedding of LTL in PVS

Previous formalisations of the Linear Temporal Logic in PVS include Pnueli's formalisation [32] and PVS-nasalib LTL library [28]. We can categorise the first

approach as a shallow embedding of LTL into PVS, and the second one as a deep embedding, following the distinction drawn in [7,33]. A shallow embedding of a modal logic in PVS is a direct syntactic transformation of the logic into PVS, and it relies PVS's existing syntax and type system. On the other hand, a deep embedding defines an abstract datatype that models the syntax, and defines functions that operate on this datatype by recursion and case analysis. A deep embedding is to be preferred for proving properties *about* the logic at issue, whilst a shallow embedding should be preferred for proving properties stated *in* the logic. After some experimentation with the LTL library from [28] we realized that a shallow embedding of LTL in PVS is more efficient for our purpose. Our formalisation follows the style of the elementary shallow embedding of propositional modal logic in PVS from [33].

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[1] 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 [36].

4.3 Certifying Proofs for Invariants: a Basic PVS Strategy

We are interested in the invariant checking problem $\mathcal{M} \models \mathbf{G}(\phi)$, where $\mathcal{M} = \langle X, \mathcal{I}, T \rangle$ and ϕ is a propositional formula defined over X. Following the approach from [24], in order to deal with this problem we provide a proof for

$$\mathcal{I} \wedge \mathbf{G}(T) \rightarrow \mathbf{G}(\phi) \qquad (1)$$

[1] 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`.

Given a propositional formula ϕ, we say it is inductive w.r.t. the transition system if ϕ holds at the initial state, and ϕ holds in all states reachable from the states that satisfy ϕ. For these cases, a proof of 1 can be obtained using the LTL-induction rule [23]:

$$\frac{\phi \qquad \mathbf{G}(\phi \rightarrow \mathbf{X}(\phi))}{\mathbf{G}(\phi)}\ \mathrm{I}$$

One proves that $\mathcal{I} \rightarrow \phi$ and $\mathbf{G}(T) \rightarrow \mathbf{G}(\phi \rightarrow \mathbf{X}(\phi))$ so to conclude 1.

This induction rule has been formalised and proved as a `shallow_ltl` lemma of the theory `lemmas_shallow_ltl` [36]. We use this lemma within a PVS strategy for proving inductive invariants. The PVS proof tree starts with $\vdash valid(\mathcal{I} \wedge \mathbf{G}(T) \rightarrow \mathbf{G}(\phi))$ and applies PVS primitive rules as well as proved lemmas. The full strategy can be found at [36](`inductive_invariant_shallow_ltl_macro`).

However in most cases the invariant property to be proved ϕ is not necessarily inductive w.r.t. $\mathcal{M}$. Most model checkers prove these cases by generating a formula ψ that is inductive and that is stronger than ϕ, i.e. such that $\psi \rightarrow \phi$. To certify these cases, we developed a PVS strategy that takes an inductive invariant ψ as an explicit parameter (provided by the model checking phase) and uses the previously described strategy for inductive invariant as subroutine. The goal is again to provide a proof for the validity of 1. In our strategy we make of use the PVS rule `Case` which allows us to assume a formula and subsequently prove this formula to be true [35]. Thus with `Case` we assume that *i)* $\mathcal{I} \wedge \mathbf{G}(T) \rightarrow \mathbf{G}(\psi)$ is valid, and that *ii)* $\psi \rightarrow \phi$ is globally valid. Hence the strategy uses a proved lemma to infer that 1 follows from these assumptions. This concludes the *structural part* of the proof. Subsequently, it proves assumption *i* by using the strategy for proving inductive invariant, and assumption *ii* using the PVS built-in SAT solver Yices. This concludes the *proof obligations discharging part* of the proof. The full strategy can be found at [36] under the name of `general_invariant_shallow_ltl_macro` as well as an example of its application and the resulting proof for 1 applied to specific formulae for $\mathcal{I}$, T and property ϕ. This also provides an example of applying the strategy for proving inductive invariants, as the last is used as a subroutine of the former.

4.4 Phase Abstraction as a Temporal Deductive Rule

As discussed in Section 3.3, phase abstraction is a transformation technique that partitions the system according to identified phases. We consider two-phase abstraction, which is when the system partitions into two equivalence classes: the sets of states reachable over even and over odd time-frames. We show how phase abstraction can be formalised as a rule in LTL, and therefore the traditional proof for invariant based on the generation of an inductive invariant (discussed in Section 4.3) can be extended to take into account this transformation of the original system. In this way, we extend the set of LTL rules capturing transformation techniques presented in [24].

Let us define the following formulae:

$$\mathcal{I} \to \hat{\chi} \tag{2}$$

$$\mathbf{G}T \to \mathbf{G}(\hat{\chi} \to \mathbf{X}(\mathbf{X}(\hat{\chi})) \tag{3}$$

$$\mathbf{G}(\hat{\chi} \to \phi) \wedge \mathbf{G}((\hat{\chi} \wedge T) \to \mathbf{X}(\phi)) \tag{4}$$

We model phase abstraction as an LTL rule as follows, where ϕ is the invariant property of interest:

$$\frac{\mathcal{I} \to \hat{\chi} \qquad \mathbf{G}T \to \mathbf{G}(\hat{\chi} \to \mathbf{X}(\mathbf{X}(\hat{\chi})) \qquad \mathbf{G}(\hat{\chi} \to \phi) \wedge \mathbf{G}((\hat{\chi} \wedge T) \to \mathbf{X}(\phi))}{\mathcal{I} \wedge \mathbf{G}(T) \to \mathbf{G}(\phi)}\ \text{P-A}$$

Here, the formula $\hat{\chi}$ represents an inductive invariant of the phase-abstracted model, which is the model made of all states reachable over even time-frames. Premise 2 expresses that the initial condition implies $\hat{\chi}$. Premise 3 expresses that, starting from a state where $\hat{\chi}$ is true and following the transition relation, $\hat{\chi}$ holds every two steps of the original model. This makes $\hat{\chi}$ an inductive invariant of the phase-abstracted model. Finally Premise 4 expresses that $\hat{\chi}$ is not just *any* invariant of the phase-abstracted model, but it is an invariant suitable to grant a proof of the original property ϕ: ϕ holds whenever $\hat{\chi}$ holds, and starting from a state where $\hat{\chi}$ holds and looking ahead of one step, ϕ holds at the next state too. Our job is now to provide a proof of correctness of the P-A rule, so that this rule can safely be used in proofs for invariants.

In order to prove that the conclusion of our P-A rule follows from its premises we reason as follows. We define $\alpha \coloneqq \hat{\chi} \vee \mathbf{X}(\hat{\chi})$. We prove that, under P-A's premises 2 and 3, α is an invariant of the original system, i.e. that $\mathcal{I} \wedge \mathbf{G}(T) \to \mathbf{G}(\alpha)$. This amounts to prove the following auxiliary lemma.

Lemma 1. $(2) \wedge (3) \to (\mathcal{I} \wedge \mathbf{G}(T) \to \mathbf{G}(\alpha))$.

The lemma above has been formalised and proved in PVS. We report a schema of the PVS proof, and we refer to [36] for the fine-grained proof.

$$\dfrac{\dfrac{\dfrac{(2)}{\mathcal{I} \to \alpha} \qquad \dfrac{(3)}{\mathbf{G}(T) \to \mathbf{G}(\alpha \to \mathbf{X}(\alpha))}}{\mathcal{I} \wedge \mathbf{G}(T) \to (\alpha \wedge \mathbf{G}(\alpha \to \mathbf{X}(\alpha)))}}{\mathcal{I} \wedge \mathbf{G}(T) \to \mathbf{G}(\alpha)}\ \text{I}$$

Finally we prove that under premise 4 and $\mathbf{G}(\alpha)$ we have $\mathbf{G}(\phi)$. Hence we conclude that $\mathcal{I} \wedge \mathbf{G}(T) \to \mathbf{G}(\phi)$.

The full formalisation and proof of the P-A rule has been done in PVS in our theory `lemmas_shallow_ltl`. The schema of the proof is as follows, and we refer to [36] for the fine-grained proof.

$$\dfrac{\dfrac{\dfrac{(2) \quad (3)}{\mathcal{I} \wedge \mathbf{G}(T) \rightarrow \mathbf{G}(\alpha)}\text{Lemma 1} \quad [\mathcal{I} \wedge \mathbf{G}(T)]}{\mathbf{G}(\alpha)} \quad (4) \quad [\mathbf{G}(T)]}{\dfrac{\mathbf{G}(\phi)}{\mathcal{I} \wedge \mathbf{G}(T) \rightarrow \mathbf{G}(\phi)}}$$

We intend to generalize such a proof to $k \neq 2$ as a future work, where k is the number on identified phases.

A PVS Strategy for Phase Abstraction We propose a PVS strategy that modifies the basic strategy from Section 4.3 to take into account phase abstraction. The PVS proof tree starts with the goal: $\vdash \mathit{valid}(\mathcal{I} \wedge \mathbf{G}(T) \rightarrow \mathbf{G}(\phi))$, where $\mathcal{I}$, T and ϕ are specific formulae describing the transition system at issue and the property we need to verify. The strategy takes the formula $\hat{\chi}$, which is the inductive invariant of the phase-abstracted model, as an explicit parameter. This formula is provided by the model checking phase. We initially use the rule `Case` to introduce the three premise of P-A rule 4.4 in the sequent's assumptions. Then we apply the proved PVS lemma for phase abstraction, which formalises the rule P-A, and by some manipulation using PVS rules, we conclude the goal. This concludes the *structural part* of the proof. The rule `Case` is a branching rule and requires us to verify the truth of our assumed formulae on the other proof-branch. The developed strategy proves formulae 2, 3 and 4 for the assumed values of $\mathcal{I}$, T and $\hat{\chi}$ by applying PVS rules and expanding definitions from `shallow_ltl`, and finally invoking the Yices SAT solver. This concludes the *proof obligations discharging part* of the proof. The full strategy can be found at [36] under the name of `phase_abstraction_kis2` as well as an example of its application and the resulting proof.

4.5 Temporal Decomposition as a Temporal Deductive Rule

When using temporal decomposition (Section 3.3), the initial condition $\mathcal{I}$ is replaced by a condition ι that overapproximates the set of reachable states after k transitions. In [24] it is shown that this technique can be formalised as a derived LTL rule. In the following $\mathbf{X}^0(P) := P$ and $\mathbf{X}^{n+1}(P) := \mathbf{X}(\mathbf{X}^n(P))$ for all $n \geq 0$.

$$\frac{(\mathcal{I} \wedge \bigwedge_{0 \leq i < k} (\mathbf{X}^i(T))) \rightarrow \mathbf{X}^k(\iota) \quad (\mathcal{I} \wedge \bigwedge_{0 \leq i < k} (\mathbf{X}^i(T))) \rightarrow (\bigwedge_{0 \leq i < k} (\mathbf{X}^i(\phi))) \quad \mathbf{G}(\iota \wedge \mathbf{G}(T) \rightarrow \mathbf{G}(\phi))}{\mathcal{I} \wedge \mathbf{G}(T) \rightarrow \mathbf{G}(\phi)} \text{ T-D}$$

The first premise models that after k steps the condition ι is reached. The third premise models that once we are in a state of $\mathcal{M}$ where ι holds, ϕ will hold thereafter. The second premise checks that ϕ holds in the first $k-1$ steps, as for ϕ to be an invariant of original transition system it has to hold *before* ι too. The rule concludes that ϕ is an invariant of the transition system.

We have formalised and proved the T-D rule as a PVS lemma [36] within our theory `lemmas_shallow_ltl`. This enabled us to develop a PVS strategy that takes into account temporal decomposition when proving invariant properties.

A PVS Strategy for Temporal Decomposition The PVS proof tree starts with the goal: $\vdash valid(\mathcal{I} \wedge \mathbf{G}(T) \rightarrow \mathbf{G}(\phi))$. The strategy takes the formula ι and the natural number k as explicit parameters. They are provided by the model checking phase. It uses the rule `Case` to introduce the three premises of T-D rule 4.5 in the sequent's assumptions. It then applies the proved PVS lemma for temporal decomposition, and by some manipulation it concludes the goal. This concludes the *structural part* of the proof. The rule `Case` is a branching rule and it requires us to verify the truth of our assumed formulae (i.e. the T-D rule's premises) on the other proof-branches. The first two premises of T-D have to be established by applying PVS rules and definitions from `shallow_ltl` and the Yices SAT solver. The third premise is an invariant claim on the temporally decomposed model $\mathcal{M}^{'} = \langle \iota, T \rangle$. So, the currently described strategy calls a version of `general_invariant_shallow_ltl_macro` (see Section 4.3) as a subroutine to complete the proof of this branch. This concludes the *proof obligations discharging part* of the proof. The full strategy can be found at [36] under the name `tmp_decomposition` as well as an example of its application and the resulting proof.

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 of two model checking tools, nuXmv [15] and PdTRAV [13]. Both tools take as input a model in Aiger [4] format, and produce inductive invariants as Aiger combinational circuits expressed over the state variables of the model.[2] We then apply a simple Python script to translate the input model and the generated invariant into a PVS theory. We make the script available (together with the rest of the our toolchain) at [36]. The Python script is relatively simple, and it can be verified through standard software verification methodologies.

We could use some certified toolchain in a future implementation to verify also these steps.

[2] In the case of temporal decomposition and phase abstraction, such invariants are over state variables *after* the model transformations have been applied. Moreover, in the case of temporal decomposition, the model checkers also output the number k of applied temporal decomposition steps. In case of generalized phase abstraction, with $k \neq 2$, such a value would have to be an output of the model checker as well.

Benchmark Set For our evaluation, we have collected a total of 105 distinct problem instances from different families, stemming from previous Hardware Model Checking Competitions (HWMCC) [2].

In order to make it possible to properly and univocally keep track of all the elements comprising the circuits under verification, we have incorporated an explicit symbol table in all chosen base model instances. Each symbol table labels the inputs, outputs, and state elements of each circuit. Such a step does not alter the nature of the benchmark at all, no elements are removed: it is just a matter of providing names to each of the circuit components.

For each of the selected benchmarks, we generated intermediate representations of the models, including possible simplification steps, if any, that were applied during the process. All benchmarks, scripts, intermediate representations, certificates, and run logs have been documented and made available for reference [36].

The set of selected benchmarks has been identified by choosing UNSAT instances that could be verified and subsequently for which an invariant could be derived, without any form of simplification and within a given time limit of 1800 seconds. We intend to expand the experimental evaluation further, as we include additional simplification strategies in the future, in order to cover a larger set of available benchmarks and to tackle harder instances.

Results In order to evaluate our proposed approach, we implemented the described methodology within different model checking suites.

The experiments 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. All jobs were managed by SLURM

with a memory limit of 4 GB and a wall-time limit of 1800 seconds per experimental run.

In order to keep verification and certification steps streamlined we selected a single strategy to perform the verification phase.

Specifically, we utilize our IC3 engine [9] without introducing any additional simplification steps. This approach is designed to produce a specialized invariant, which serves as a certificate. Invariants were derived from the base models, as well as from runs where either temporal decomposition or phase abstraction was used.

As a future work, we would like to extend the experimental evaluation and the set of techniques covered in order to support additional engines, such as interpolation and IGR [10], but in order to do so additional transformation techniques and simplifications steps may have to be accounted for first, as interpolants often requires additional pre-processing to enhance their usability [11, 12]. Concerning nuXmv, it has been run in the same setup as described before, taking into account only its IC3 engine.

PVS experiments were conducted using PVS8.0[3] on the set of benchmarks chosen as explained 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.

For each run we are reporting the verification time, during which the corresponding invariant/certificate has been generated as an auxiliary artifact as well as certification times for the theorem prover approach. PVS certification results are given considering type-checking times, SAT solver times and overall certification times.

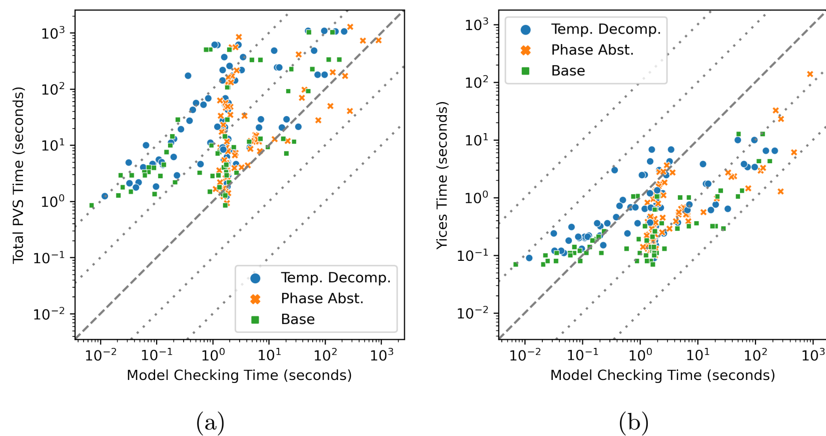

Fig. 1: Comparison between model checking verification and PVS certification times, considering overall certification times (a) and just SAT solver times (b).

Figure 1 presents preliminary results comparing the certification times of our theorem prover-based approach against model checking instances verification and invariants creation times. Figure 1a accounts for all the PVS certification contributions whereas Figure 1b focuses on just Yices SAT solver times.

Comparing original benchmarks and invariants size in terms of number of AND-gates in their circuit based representation, shows that invariants tend to be between one and two order of magnitude smaller than the original benchmarks, as desirable.

Whilst these results demonstrate the feasibility of theorem prover-based certification, 166 out of 189 runs in total were successfully proved by PVS within the allotted time limits, they also reveal a significant performance gap between the two.

---

[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.

To better understand the reasons behind this performance gap, we further analysed the breakdown of execution times for different operations within PVS. We considered all runs, without distinguishing between phase abstraction runs, temporal decomposition runs and runs with no transformations. 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 spent on calls to the SAT solver Yices.

This breakdown highlights that the theorem prover spends the majority of its time on type-checking, and on proof-checking operations other than the calls to Yices, and as shown in Figure 1b, the time spent on Yices calls is usually inferior than the model checking time. To better distinguish the contribution of each series, a logarithmic scale has been used for the $y-$axis.

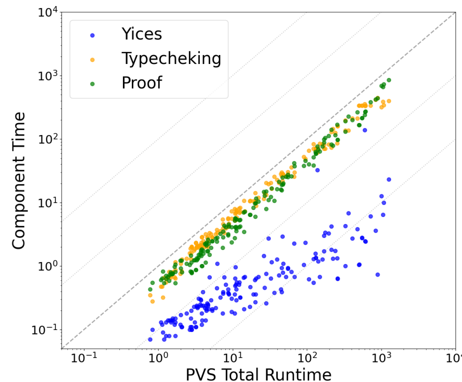

Fig. 2: Times breakdown for PVS-based runs.

Notably, the high type-checking times stem from PVS support for a rich variety of theories, which necessitates complex checks to ensure type correctness before the proof process even begins. Additionally, in our experiments, the high proof-checking times appeared to be driven more by internal bookkeeping operations and installing definitions rather than the execution of actual reasoning steps.

In conclusion, we attribute this performance gap primarily to internal bottlenecks within PVS. These inefficiencies could potentially be mitigated by an optimized, ad-hoc implementation of the theorem prover, or by engaging with the PVS maintainers to explore optimisation opportunities for our use case, or yet by adopting an alternative theorem prover with stronger support for automated proof checking rather than interactive use (LEAN [17] represents an alternative, but further exploration is needed to verify its comparative advantages). In particular, we believe that an optimized ad-hoc theorem prover could yield

proof-checking times comparable to our SAT-based approach. We leave further investigation of these possibilities for future work. Nonetheless, our prototype PVS implementation and preliminary results confirm both the soundness and feasibility of our theorem prover-based approach, which can be implemented in any off-the-shelf or custom theorem prover with LTL reasoning capabilities and that requires minimal modifications to existing model checkers. Such a possibility is one of the future steps we intend to investigate as a future work.

Limitations In this section we describe the current limitations of our prototype implementation, and the planned work to overcome these limitations.

Our certification process as it described in Section 4.1 currently applies transformations in isolation, but the framework provides proofs that are easily composed. This makes it possible to develop strategies that generate proofs combining multiple simplifications, such as composing phase abstraction with temporal decomposition. Proof composition among strategies is already implemented to some extent, as most strategies call the strategy for proving inductive invariants as their subroutine.

We also aim to strengthen the confidence provided by our certification process. As explained in Section 4.1, the proof obligation discharge component relies on the PVS SAT solver Yices. To increase trust in this fundamental part of the proof, it would be beneficial to use proof producing SAT solvers that operate with greater transparency rather than functioning as black boxes.

Finally, we already discussed in Section 5 the main bottlenecks observed with PVS performance and how we plan to overcome these.

6 Conclusions and Future Work

In this paper, we have presented a novel approach for certifying model checking results by exploiting a theorem prover and a theory of temporal deductive rules that can support various kinds of transformations and simplifications of the original system. We established the correctness of the model checking transformation algorithms using a theorem prover. We employed the resulting rules to provide proof certificates of the success of the model checker, thereby reducing the burden of model checker developers to produce these certificates themselves. In fact, our technique can be easily implemented on top of existing model checkers with minimal modifications.

Our experiments confirm both the soundness and feasibility of our theorem prover-based approach, but at the same time highlights the necessity to better integrate such an approach with existing theorem provers, to overcome practical limitations to their applicability.

In this paper, we presented our work on some simplification techniques, and we first tackled the problem of considering these simplifications "in isolation from each other". However, the framework provides proofs that are easily composed, and so it is a very feasible next step to consider the generation of proof that combine multiple simplifications.

We see several directions for future work, such as extending the proposed approach to include a more general take on phase abstraction, being able to support additional kind of simplifications, as well as extending the model checking strategies to derive invariants. We would also like to extend the theorem prover based certification beyond the invariant model checking problem to liveness and LTL model checking. For example, given that the k-liveness algorithm [16] has been formalised as a temporal deductive rule in [24], a natural extension of our work is to use the theorem prover based approach to formalise and prove k-liveness and implement a strategy for it. Moreover the same certifying model checking approach can be extended to other LTL model checking algorithms such as FAIR [8] and liveness-to-safety [3].

Acknowledgements. The authors would like to acknowledge the invaluable help extended by Dr. Natarajan Shankar and the whole PVS development team, for their support and insightful discussions concerning the PVS tool.

This work was supported in part by the European Union - Next Generation EU under the Italian National Recovery and Resilience Plan (NRRP), Mission 4, Component 2, Investment 1.3, CUP E13C22001870001, partnership on "Telecommunications of the Future" (PE00000001 - program "RESTART"). This work was also supported in part by SRC contract 2024-CT-3280.

A. Griggio, M. Roveri, G. Sindoni and S. Tonetta acknowledge the support of the PNRR project FAIR - Future AI Research (PE00000013), under the NRRP MUR program funded by the NextGenerationEU.

M. Roveri is also partially supported by the project MUR PRIN 2020 - RIPER - Resilient AI-Based Self-Programming and Strategic Reasoning - CUP E63C22000400001, and by the European Union under Horizon Europe Programme - Grant Agreement 101070537 — CrossCon.

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

References

1. Jason Baumgartner, Tamir Heyman, Vigyan Singhal, and Adnan Aziz. An abstraction algorithm for the verification of level-sensitive latch-based netlists. *Form. Methods Syst. Des.*, 23(1):39–65, July 2003.
2. A. Biere and T. Jussila. The Model Checking Competition Web Page, http://fmv.jku.at/hwmcc.
3. Armin Biere, Cyrille Artho, and Viktor Schuppan. Liveness checking as safety checking. *Electronic Notes in Theoretical Computer Science*, 66(2):160–177, 2002.
4. 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.
5. 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.

6. Per Bjesse and James Kukula. Automatic generalized phase abstraction for formal verification. In *ICCAD-2005. IEEE/ACM International Conference on Computer-Aided Design, 2005.*, pages 1076–1082. IEEE, 2005.
7. 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.
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. G. Cabodi, P. E. Camurati, A. Mishchenko, M. Palena, and P. Pasini. SAT solver management strategies in IC3: an experimental approach. *Form. Methods Syst. Des.*, 50(1):39–74, March 2017.
10. Gianpiero Cabodi, Paolo E Camurati, Marco Palena, and Paolo Pasini. Interpolation with guided refinement: revisiting incrementality in SAT-based unbounded model checking. *Formal Methods in System Design*, 60(2):117–146, 2022.
11. Gianpiero Cabodi, PE Camurati, Marco Palena, Paolo Pasini, and Danilo Vendraminetto. Logic synthesis for interpolant circuit compaction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(2):380–384, 2018.
12. Gianpiero Cabodi, PE Camurati, Marco Palena, Paolo Pasini, and Danilo Vendraminetto. Reducing interpolant circuit size through SAT-based weakening. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(7):1524–1531, 2019.
13. Gianpiero Cabodi, Sergio Nocco, and Stefano Quer. Benchmarking a model checker for algorithmic improvements and tuning for performance. *Form. Methods Syst. Des.*, 39(2):205–227, October 2011.
14. Michael L Case, Hari Mony, Jason Baumgartner, and Robert Kanzelman. Enhanced verification by temporal decomposition. In *2009 Formal Methods in Computer-Aided Design*, pages 17–24. IEEE, 2009.
15. 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.
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. Leonardo De Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*, pages 378–388. Springer, 2015.
18. The LeanSAT Developers. LeanSAT.
19. 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.
20. Nils Froleyks, Emily Yu, Armin Biere, and Keijo Heljanko. Certifying phase abstraction. In *International Joint Conference on Automated Reasoning*, pages 284–303. Springer, 2024.
21. Siddhartha Gadgil and Anand Rao. Saturn: Experiments with SAT solvers with proofs in Lean 4, 2025. Accessed: May 19, 2025.

22. Gerhard Gentzen. Untersuchungen über das logische schließen. i. *Mathematische zeitschrift*, 35, 1935.
23. Alberto Griggio, Marco Roveri, and Stefano Tonetta. Certifying proofs for LTL model checking. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–9. IEEE, 2018.
24. Alberto Griggio, Marco Roveri, and Stefano Tonetta. Certifying proofs for SAT-based model checking. *Formal Methods in System Design*, 57(2):178–210, 2021.
25. Abdalrhman Mohamed, Tomaz Mascarenhas, Harun Khan, Haniel Barbosa, Andrew Reynolds, Yicheng Qian, Cesare Tinelli, and Clark Barrett. Lean-SMT: An SMT tactic for discharging proof goals in Lean. In *Proceedings of the 37th International Conference on Computer Aided Verification (CAV 2025)*, 2025.
26. César A Munoz. Batch proving and proof scripting in PVS. Technical report, National Institute of Aerospace, 2007.
27. 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.
28. NASA. PVS-nasalib LTL library. `https://github.com/nasa/pvslib/tree/master/LTL`. Accessed: 2025-02-21.
29. Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In *International Conference on Automated Deduction*, pages 748–752. Springer, 1992.
30. 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.
31. Amir Pnueli. The temporal logic of programs. In *18th annual symposium on foundations of computer science (SFCS 1977)*, pages 46–57. IEEE, 1977.
32. Amir Pnueli and Tamarah Arons. TLPVS: a PVS-based LTL verification system. In *Verification: Theory and Practice: Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, pages 598–625. Springer, 2003.
33. John Rushby. PVS embeddings of propositional and quantified modal logic. *arXiv preprint arXiv:2205.06391*, 2022.
34. Carl-Johan H. Seger and Randal E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Form. Methods Syst. Des.*, 6(2):147–189, March 1995.
35. 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.
36. Giulia Sindoni, Paolo Pasini, Gianpiero Cabodi, Paolo E. Camurati, Alberto Griggio, Paolo Palena, Marco Roveri, and Stefano Tonetta. Fbk-pdt-cert25. `https://gitlab.fbk.eu/gsindoni/fbk_pdt_cert25`. Accessed: 2025-02-24.
37. 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.
38. Emily Yu, Nils Froleyks, Armin Biere, and Keijo Heljanko. Towards compositional hardware model checking certification. In *2023 Formal Methods in Computer-Aided Design (FMCAD)*, pages 1–11. IEEE, 2023.