From Three-Valued Model Checking to Temporal Proofs

Abstract. Model checking automatically verifies whether a model of the system under analysis satisfies a property of interest. It returns *true* if the property is satisfied, *false* and a violating behavior if it is not. Three-valued model checking has been proposed to support verification when some portions of the model are unspecified. Differently from the classical two-valued case, it also returns a *maybe* value and a possibly violating behavior if the property satisfaction depends on how the unspecified parts are refined.
When no violating or possibly violating behavior is returned, model checking does not usually provide the designer with any details about the reasons *why* the property is satisfied. This is the purpose of deductive verification. Deductive verification frameworks produce a formal proof of the property satisfaction, but have only been studied for the two-valued case. This paper proposes THRIVE, a unified approach that enriches three-valued model checking with a deductive verification engine that generates proofs which explain why a *true* or *maybe* result is returned.

1 Introduction

Several multi-valued model checking techniques, such as [4, 5, 14, 6], have been proposed to support the verification of models that are *incomplete*. A model is incomplete whenever its state space is not fully specified. Three-valued model checking extends classical two-valued model checking by returning an additional *maybe* value whenever the property satisfaction depends on how the incompleteness is removed. More precisely, a three-valued model checking algorithm returns one of the following values: *true*, if the property ϕ *definitely holds* (it is satisfied regardless of how incomplete parts are refined), *false* if ϕ *does not hold* (a violating behavior which does not depend on the incomplete parts exists) and *maybe* if ϕ *possibly holds* (its satisfaction depends on the parts to be refined).

In the classical context of two-valued model checking, algorithms do not provide the designer with enough information. Although a sample violating behavior is normally returned when the property is violated, no equally useful insight is provided if the property is proved to hold. Indeed, in this case, it might be useful to receive a formal explanation of the reason *why* the system satisfies the property. To achieve this goal, the verification framework can be equipped with a deductive verification engine that formally justifies why the model checking procedure has failed in the search of a counterexample by exploiting the state space generated during the search. Deductive verification algorithms have been developed for fully specified models [22, 21], but no known similar approach deals

with incomplete models. This is a remarkable limitation in practical contexts based on incremental development, where the designer may start by providing an initial, high-level version of the model, which is iteratively narrowed down as design progresses and uncertainties are removed. Whenever the verification answer is true or maybe, the proof can be used by the designer as a guidance in the refinement process, and as a confirmation on the correctness of the design choices already performed. In some cases, it may even suggest that actually the property does not capture the intended correctness condition, and it should be modified. For this reason the integration of deductive verification techniques in a multi-valued context can guide the designer towards a correct development.

This paper proposes THRIVE, a THRee valued Integrated Verification Engine for incomplete models. THRIVE enriches model checking for incomplete models with a deductive verification engine that justifies whether the verified system definitely satisfies or possibly satisfies the property of interest. Whenever the property *definitely holds*, the deductive verification approach proves that neither a definitely violating nor a possibly violating behavior can be found in the current instance of the model. If the property *possibly holds*, the model checker returns a possible counterexample, which describes a possible behavior that violates the property. In addition, the deductive verification engine returns to the designer a proof that specifies why a violating behavior cannot be found in the current model. Finally, whenever the property *does not hold*, a counterexample which specifies a violating behavior is returned.

We consider the model M and the property ϕ expressed respectively as a Partial Kripke Structure (PKS) [4] and a Linear Temporal Logic (LTL) [23] formula, but the proposed framework can be also extended to support other formalisms. The framework is founded on previous theoretical results [5, 10, 12, 22], which have been modified to fit into the proposed approach. It exploits the formal definition of the three-valued semantics of LTL formulae over infinite words presented in [12] and uses a model checking algorithm—in line with the one presented in [5, 14]. The model checking algorithm is equipped with a customization of the deductive verification framework proposed in [22]. We discuss the applicability of the proposed integrated verification environment considering the three-valued [4] and the thorough [5] LTL semantics. Furthermore, we also analyze the validity of THRIVE over a particular subset of LTL formulae called self-minimizing [10] LTL formulae.

The benefits of the approach are evaluated using a simple semaphore example. The model of the semaphore is obtained by abstracting the one presented in [2] and is considered w.r.t. three LTL properties. THRIVE helps the designer in understanding which choices of component design lead to (in)correct behavior of the system, and why some choices imply that a property definitely/possibly holds. This information provides the designer with good insights on how to refine/modify the model of the system.

The rest of the paper is organized as follows. Section 2 introduces the semaphore example. Section 3 presents the modeling formalisms, their properties and verification procedures upon which this work is based. Section 4 describes

THRIVE. Section 5 evaluates the approach on the semaphore example. Section 6 discusses the applicability of the approach. Section 7 presents the related works. Finally, Section 8 concludes the paper.

2 Motivating Example.

Let us consider the design of a grade crossing semaphore. For illustration purposes we assume the designer has identified three simple requirements. Requirement ϕ_1 specifies that *red* lights up infinitely often ($\Box\Diamond red$), which forces cars to stop when the train approaches the crossing. Requirement ϕ_2 states that *green* lights up infinitely often ($\Box\Diamond green$), which allows the cars to traverse the crossing. Finally, requirement ϕ_3 specifies that always, after *red*, *green* is permanently on ($\Box(red \rightarrow \Box green)$). Note that ϕ_3 is wrong (i.e., it does not capture desirable behaviors in the current context). Nevertheless it is used for illustration purposes.

Starting from this specification, suppose that the designer proposes the incompletely specified model of the semaphore shown in Figure 1. Each state is associated with the values of the propositions g and r (denoting *green* and *red*) holding in that state, which specify whether the green and the red lights are on or off in that state. For example, in state s_0 the red light is on ($r = \top$) while the green is off ($g = \bot$). Instead, s_2 is a state to which the semaphore may be brought, for instance by a manual command. The designer still has to choose whether, in this state, the green and red lights should be on or off. This is indicated by associating the value ? to the propositions g and r. The designer might refine the model by setting to either $\top$ or $\bot$ the g/r proposition in s_2.

Regardless of the temporary incompleteness, the designer wants to evaluate the satisfaction of the properties of interest. By model checking, it is possible to verify that the model does not contain any behavior that violates or possibly violates ϕ_1. However, the result does not provide any information on why ϕ_1 is satisfied. Instead, a model checker equipped with a deductive verification framework would reveal that the system enters infinitely often state s_0, in which the red light is turned on, thus explaining why property ϕ_1 is satisfied.

Let us now consider property ϕ_2, which possibly holds. The model checking algorithm returns a *possible* counterexample: by turning off the green light in state s_2, the system may infinitely loop over s_0 and s_2 without g ever being true. However, the search of a *definitive* counterexample has failed. The deductive verification framework would explain to the designer why this search failed. As a

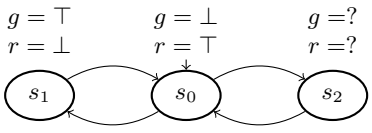

Fig. 1. System model M

matter of fact, if a counterexample is present, it must be present in any possible completion of the model, i.e., for any possible assignment to the light in the state s_2. But, let us consider a possible refinement of the incomplete state s_2 where the green light is on. Since the green light is on in s_1 (s_2), it is possible to deduce that the light is eventually green in state s_1 (s_2). Since it is allowed to loop over states s_0, s_1 and s_2, it is possible to deduce that "always eventually green" is true in s_0, s_1 and s_2. Thus, ϕ_2 is satisfied in s_0, the initial state. In conclusion, when the model checker returns maybe, it outputs a possible counterexample for a possible refinement of the incomplete states. In addition, the deductive verification framework would explain that the search of a definitive counterexample has failed in the case in which the green light is on in s_2. This provides a useful insight to the designer, explaining that, by setting g to $\top$ in s_2, one obtains a model satisfying ϕ_2 while, in the opposite case, ϕ_2 is violated.

Finally, when property ϕ_3 is considered, the model checking algorithm returns a definitive counterexample since ϕ_3 is not satisfied, regardless of further refinements of s_2. Indeed, there exists an infinite run among states s_0 and s_1 in which a red light is not followed by a permanently green light. Indeed, the requirement that the designer probably meant to specify is $\Box(red \rightarrow \bigcirc green)$.

3 Background

This section presents the modeling formalisms, their properties and verification algorithms upon which THRIVE is based. The section first briefly overviews how functional properties are checked over complete models. Then it discusses how incomplete models are verified. Finally, it describes how the deductive verification framework exploits the model checking procedure of complete models to generate a proof that explains why a property is satisfied. An extended version of the paper, that includes all the formal details, can be found at `goo.gl/g3tmzG`.

3.1 Checking complete models.

Given a Kripke Structure M (KS) [17], the model checking procedure verifies whether a Linear Temporal Logic (LTL) [23] formula ϕ holds or does not hold in M. The procedure works in three steps [8]: 1. generation of a non-deterministic Büchi automaton $\mathcal{A}_{\neg\phi}$ from an LTL formula ϕ; 2. generation of the product $\mathcal{G} = M \otimes \mathcal{A}_{\neg\phi}$; 3. emptiness check of $\mathcal{G}$.

3.2 Checking incomplete models.

We present Partial Kripke Structures [4] (PKSs), which is the formalism we assume the designer has used to describe the system under development. We discuss the three-valued [4] and thorough [5] semantics of LTL formulae over PKSs, since the knowledge of these two semantics is necessary to understand when THRIVE can be applied. Finally, we introduce the verification procedure

of an LTL formula over a PKS for the three-valued semantics, which is the base block upon which our deductive verification procedure is constructed.

Partial Kripke Structures (PKSs) extend KSs by allowing a proposition in a given state to be labelled with ?, to represent an unknown value of a proposition.

Definition 1 (Partial Kripke Structure [4] (PKS)). *A PKS M is a tuple $\langle S, R, S_0, AP, L\rangle$, where: S is a set of* states*; $R \subseteq S \times S$ is a* left-total transition relation *on S; S_0 is a set of initial states; AP is a set of atomic propositions; $L : S \times AP \to \{\top, ?, \bot\}$ is a* function *that, for each state in S, associates a truth value in the set $\{\top, ?, \bot\}$ to every atomic proposition in AP.*

The grade crossing semaphore shown in Figure 1 and described in Section 2, is an example of how the system can be represented by means of a PKS.

A *completion* of a PKS M is a KS M' that completes M by assigning values to the unknown propositions. We will use $\mathcal{C}(M)$ to indicate all the completions of a PKS M.

Two different semantics of LTL can be considered: the three-valued or the thorough semantics.

The *three-valued LTL semantics* specifies that a formula ϕ definitely holds in a PKS M if it is true for all possible values of the unknown propositions in M. Likewise, it is definitely violated if it is false despite the unknown values.

Definition 2 (Three-valued LTL semantics [12]). *Given a PKS $M = \langle S, R, S_0, AP, L\rangle$, a path $\pi = s_0, s_1, \ldots$, and a formula ϕ, the three-valued semantics $[(M, \pi) \models \phi]$ is defined inductively as follows:*

$$
\begin{array}{lcl}
[(M,\pi) \models p] & \stackrel{def}{=} & L(s_0, p) \\
[(M,\pi) \models \neg\phi] & \stackrel{def}{=} & \mathrm{comp}([(M,\pi) \models \phi]) \\
[(M,\pi) \models \phi_1 \wedge \phi_2] & \stackrel{def}{=} & \min([(M,\pi) \models \phi_1], [(M,\pi) \models \phi_2]) \\
[(M,\pi) \models \bigcirc\phi] & \stackrel{def}{=} & [(M,\pi^1) \models \phi] \\
[(M,\pi) \models \phi_1 \,\mathcal{U}\, \phi_2] & \stackrel{def}{=} & \max\limits_{j\geq 0}(\min(\{[(M,\pi^i) \models \phi_1] | i < j\} \cup \{[(M,\pi^j) \models \phi_2]\}))
\end{array}
$$

The conjunction (disjunction) is defined as the minimum (maximum) of its arguments, following the order $\bot < ? < \top$. Negation is defined by the function comp (complement), which maps $\top$ to $\bot$, $\bot$ to $\top$, and ? to ?. These functions are extended to sets considering $\min(\emptyset)=\top$ and $\max(\emptyset)=\bot$.

Given a PKS $M = \langle S, R, S_0, AP, L\rangle$, a state s, and a formula ϕ, $[(M, s) \models \phi] \stackrel{\text{def}}{=} \min(\{[(M,\pi) \models \phi] \mid \pi^0 = s\})$. Intuitively this means that, given a formula ϕ, each state s of M is associated with the minimum of the values obtained considering the LTL semantics over any path π that starts in s.

A PKS M *definitely satisfies* a property ϕ ($[M \models \phi] = \top$) iff for all initial states $s_0 \in S_0$ of M, $[(M, s_0) \models \phi] = \top$. A PKS M *does not satisfy* the property ϕ ($[M \models \phi] = \bot$) iff there exists an initial state $s_0 \in S_0$ of M such that $[(M, s_0) \models \phi] = \bot$. A PKS *possibly satisfies* ϕ otherwise. In the rest of the paper

we will use interchangeably the expressions "M definitely satisfies ϕ" and "ϕ holds in M", "M does not satisfy ϕ" and "ϕ does not hold in M", as well as "M possibly satisfies ϕ" and "ϕ possibly holds in M".

The three-valued semantics does not behave always as expected: there are cases in which ϕ possibly holds but all the KSs completions (obtained by replacing the ? values with $\top$ and $\bot$) actually satisfy (or do not satisfy) ϕ. For instance, this happens when ϕ is a tautology or it is unsatisfiable with a traditional two-valued interpretation.

The *thorough LTL semantics*—differently from the three-valued semantics—specifies that a formula is possibly satisfied only if there exist two completion KSs M_1 and M_2, that definitely satisfy and violate ϕ, respectively.

Definition 3 (Thorough semantics [5]). *Let ϕ be an LTL formula and M a PKS, the truth value of a formula ϕ of a thorough temporal logic semantics, written $[M \models \phi]_t$, is defined as follows:*

$$[M \models \phi]_t \stackrel{def}{=} \begin{cases} \top & \text{if } M' \models \phi \text{ for all } M' \in \mathcal{C}(M) \\ \bot & \text{if } M' \not\models \phi \text{ for all } M' \in \mathcal{C}(M) \\ ? & \text{otherwise} \end{cases} \tag{1}$$

Proposition 1 ([12]). *Given a PKS and an LTL formula ϕ,*

1. $[M \models \phi] = \top \Rightarrow [M \models \phi]_t = \top$.
2. $[M \models \phi] = \bot \Rightarrow [M \models \phi]_t = \bot$.

That is, a formula which is true (false) under the three-valued semantics is also true (false) under the thorough semantics.

Given the three-valued and the thorough semantics, there exists a subset of LTL formulae, indicated in literature as *self-minimizing*, such that the two semantics coincide. Formally,

Proposition 2 ([10]). *Given a model M and a self-minimizing LTL property ϕ, then $[M \models \phi] = [M \models \phi]_t$.*

In the following, we present a model checking algorithm for LTL formulae when a three-valued semantics is considered. The algorithm exploits two classical model checking procedures for KSs. These procedures consider a version of M, called *complement-closed* structure [5], in which for every proposition $p \in AP$, there exists a new proposition q, called complement-closed proposition, such that $L(s,p) = \text{comp}(L(s,q))$, for all $s \in S$. The proposition q, whose value is complementary to the one of p, is indicated as $\overline{p}$. For example, the complement-closed version of the PKS of the semaphore example is presented in Figure 2, where the atomic propositions $\overline{g}$ and $\overline{r}$ are associated with the complement of the values of propositions green (g) and red (r), respectively.

The model checking procedure for a PKS M is based on an optimistic and pessimistic approximation of M's complement-closure. The optimistic (pessimistic) approximation function L_{opt} (L_{pes}) associates the value $\top$ ($\bot$) to each atomic

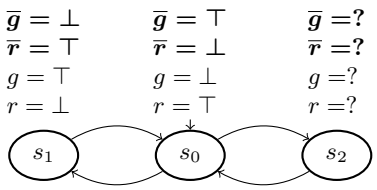

Fig. 2. The PKS of the crossing semaphore.

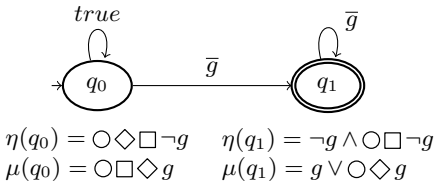

Fig. 3. The BA associated with ϕ_2.

proposition of the complement-closure of M with value ?. Thus, in the optimistic (pessimistic) approximation, the propositions g and r associated with the value ? are assigned the value $\top$ ($\bot$). Given a PKS $M = \langle S, R, S_0, L\rangle$, we have $M_{pes} = \langle S, R, S_0, L_{pes}\rangle$ for the pessimistic structure, and $M_{opt} = \langle S, R, S_0, L_{opt}\rangle$ for the optimistic one.

The three-valued model checking algorithm assumes that property ϕ is rewritten using complement-closed propositions. The rewriting procedure works in two steps. First, the formula is expressed such that negations only appear in front of atomic propositions. Second, each negated proposition is substituted by the corresponding complemented proposition.

Theorem 1 (Three-valued model checking [5][1]). *Let ϕ be an LTL formula obtained using the procedure just discussed, $M = \langle S, R, S_0, L\rangle$ a PKS with $s \in S$, and M_{pes} and M_{opt} the corresponding pessimistic and optimistic structures. Then*

$$[(M, s) \models \phi] = \begin{cases} \top & \text{if } (M_{pes}, s) \models \phi \\ \bot & \text{if } (M_{opt}, s) \not\models \phi \\ ? & \text{otherwise} \end{cases} \tag{2}$$

This technique exploits two runs of the classical two-valued model checking performed respectively on a pessimistic and an optimistic completion of the incomplete model M. Intuitively, in the construction of the optimistic completion M_{opt}, the algorithm "tries to do its best" to build a KS which satisfies ϕ. If a violating behavior is found in M_{opt}, then a definitive counterexample is returned since the property ϕ does not hold. Vice versa, in the construction of the pessimistic completion M_{pes}, the three-valued model checker tries to construct a KS which violates ϕ. If no violating behavior is found in M_{pes}, then ϕ definitely holds. Otherwise, it could be the case where there exists some completion in which ϕ holds and others in which it does not hold. This implies that a ? value is returned by the three-valued model checker. Note that whenever a $\top$ or $\bot$ value

[1] Note that Theorem 1 is stated for Positive Propositional Modal Logic in [5] but is proved to be also valid for LTL (see [5, 10, 12]).

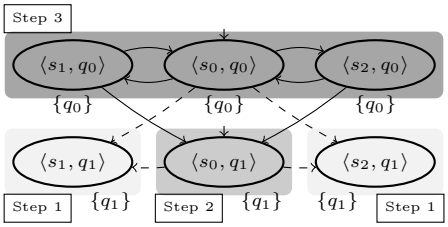

Fig. 4. Product $I_{opt} = M_{opt} \otimes \mathcal{A}_{\neg\phi_2}$

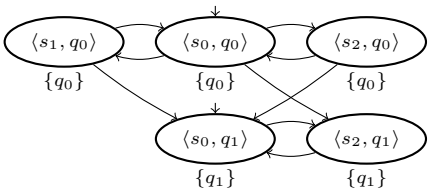

Fig. 5. Product $I_{pes} = M_{pes} \otimes \mathcal{A}_{\neg\phi_2}$

is returned, the property ϕ is guaranteed to be true/false in all the completions of M. For additional details on this procedure see [5, 12].

Properties ϕ_1, ϕ_2 and ϕ_3 of the crossing semaphore example are respectively satisfied, possibly satisfied and not satisfied for the model M of Figure 2.

Let us consider property ϕ_2. The procedure first checks if ϕ_2 is violated. The intersection I_{opt} between the optimistic structure M_{opt} (obtained by changing all the ? values in $\top$) and the BA $\mathcal{A}_{\neg\phi_2}$ associated with the property $\neg\phi_2$ (represented in Figure 3) is computed. The automaton I_{opt} is presented in Figure 4. The transitions marked with dashed-lines and the grey frames do not belong to the intersection: their meaning will be described in the following. The state space of I_{opt} is explored in search of a definitive counterexample. Since in I_{opt} no state labelled with an accepting state of $\mathcal{A}_{\neg\phi_2}$ (i.e., $\{q_1\}$) can be entered infinitely often, no counterexample is identified. Then, the procedure checks if ϕ_2 is possibly violated. The intersection I_{pes} between the pessimistic structure M_{pes} (obtained by changing all the ? values in $\bot$) and $\mathcal{A}_{\neg\phi_2}$ is computed. The automaton I_{pes} is presented in Figure 5. The state space of I_{pes} is explored in search of a possibly violating behavior. Since it exists a state labelled with an accepting state of $\mathcal{A}_{\neg\phi_2}$ (i.e., $\{q_1\}$) which can be entered infinitely often (both $\langle s_2, q_1\rangle$ and $\langle s_0, q_1\rangle$), a possible counterexample is returned. Consequently, ϕ_2 is possibly satisfied, i.e., there exists a completion M_{pes} of M that violates ϕ_2. For example, if $g = \bot$ and $r = \bot$ to the propositions in s_2, we obtain a violating behavior when the system moves alternatively between s_0 and s_2.

3.3 Deductive verification

The deductive verification framework used in this work is based on [22]. It is developed on top of the two-valued model checker described in Section 3.1. Given a stucture M and an LTL property ϕ, which is satisfied by M, the deductive verification framework produces a proof which explains why $M \models \phi$. The algorithm considers the intersection automaton $\mathcal{G} = M \otimes \mathcal{A}_{\neg\phi}$.

The approach described in the following is built on three elements. 1. every state $q \in Q$ of $\mathcal{A}_{\neg\phi}$ is associated with an LTL formula $\eta(q)$ such that, for every accepting run $\sigma = q_0, q_1, ...$ of $\mathcal{G}$, $\sigma_i \models \eta(q_i)$. This formula is computed during

the procedure that converts the LTL formula $\neg\phi$ into $\mathcal{A}_{\neg\phi}$ [9]. For instance, the state q_1 of the automaton presented in Figure 3 is associated with the formula $\eta(q_1) = \neg g \wedge \bigcirc\Box\neg g$; 2. given a state $\langle s, q\rangle$ of the automaton $M \otimes \mathcal{A}_{\neg\phi}$, the property $\eta(q)$ associated with the state q of $\mathcal{A}_{\neg\phi}$ is *not* satisfied in s. Indeed, if $\eta(q)$ was satisfied, a counterexample would be found. Thus, the negation $\mu(q)$ of $\eta(q)$ holds in s; 3. each node $\langle s, q\rangle$ which was not created during the computation of $M \otimes \mathcal{A}_{\neg\phi}$, is such that s does not satisfy $\eta(q)$, i.e., $s \models \mu(q)$. Each of these nodes, called *failed node*, causes a failure in the search of a counterexample and ensures the satisfaction of ϕ in the corresponding state of the system.

The deductive verification framework first enriches the product automaton $M \otimes \mathcal{A}_{\neg\phi}$ by considering also failed nodes as part of it. Each failed node is a node in which the search of a counterexample has failed. Thus, given a failed node $\langle t, p\rangle$, we can write the failure axiom $t \models \mu(p)$. For example, the node $\langle s_1, q_1\rangle$ of the intersection automaton presented in Figure 4 is a failed node since s_1 satisfies the property $\mu(q_1) = g \vee \bigcirc\Diamond g$ associated with the state q_1.

Then, a set of deductive rules is applied to produce the proof, specifically: 1. *Successors rule.* Given a state $\langle s, q\rangle$ of the intersection automaton, if for each of its successors $\langle s_i, q_j\rangle$ the state s_i of M satisfies the formula $\mu(q_j)$, then also s satisfies $\mu(q)$. Intuitively, the rule is based on two observations. First, each successor $\langle s_i, q_j\rangle$ of $\langle s, q\rangle$ does not cause a violation of ϕ, i.e., it ensures that $s_i \models \mu(q_j)$. Second, by moving from $\langle s, q\rangle$ to $\langle s_i, q_j\rangle$ the system has not violated the property of interest, since no counterexample was found. Thus, it must be that s satisfies $\mu(q)$. 2. *Induction rule.* It is a generalization of the successors rule applied on strongly connected components (SCCs). Given a strongly connected component $\mathcal{X}$, let us identify with $Exit(\mathcal{X})$ the set of all nodes $\langle s_i, q_j\rangle$ that do not belong to $\mathcal{X}$ and have an incoming transition from a source node in $\mathcal{X}$. If every node $\langle s_i, q_j\rangle \in Exit(\mathcal{X})$ is such that $s_i \models \mu(q_j)$, we can conclude that, for every node $\langle s, q\rangle \in \mathcal{X}$, $s \models \mu(q)$ holds. Intuitively, since all the "successors" of $\mathcal{X}$ (the nodes in $Exit(\mathcal{X})$) ensure the property satisfaction and the states in $\mathcal{X}$ do not violate the property of interest (no counterexample has been found in the intersection automaton), it must be that each state s satisfies the corresponding property $\mu(q)$. 3. *Conjunction rule.* It allows connecting any pair of conclusions made on a given state and making temporal logic interferences. All the formulae computed for a given state s are and-combined.

These rules are applied considering the partial ordering relation $\prec$ between SCCs. The relation $\mathcal{X} \prec \mathcal{X}'$ holds if there exists a transition from some state in $\mathcal{X}$ to some state in $\mathcal{X}'$. If $\mathcal{X} \prec \mathcal{X}'$, before considering the component $\mathcal{X}$, it is necessary to compute the proof of $\mathcal{X}'$.

Additional details can be found in the extended version of this paper.

4 THRIVE

This section presents THRIVE, a *THRee-valued Integrated Verification Engine*. THRIVE enriches an underlying three-valued model checker with a fully deductive verification approach.

An overview of our proposal is presented in Figure 6. THRIVE provides the designer with two kinds of information: the model checking result and the deductive verification result. The *three-valued model checking* procedure presented in Section 3 is used to verify whether the property ϕ of interest is definitely satisfied ($\top$), possibly satisfied (?) or not satisfied ($\bot$) by the current partial model. If the property is *not satisfied*, there exists some behavior which definitively violates the property of interest that does not depend on the unknown elements. The model checker returns to the designer one of these behaviors, i.e., a definitive counterexample. Whenever a property is *definitely satisfied*, its satisfaction does not depend on how the incomplete parts are refined. Finally, if the property is *possibly satisfied*, its satisfaction depends on how the uncertainty is removed. The model checker returns a possible violating behavior.

The *deductive verification* engine is executed when a $\top$ or ? value is returned by the model checker. It is used to compute a proof which specifies explicitly why the property ϕ is definitely (possibly) satisfied by M, by incrementally adding logic assertions to the framework. When the automatic procedure is completed, the user obtains a proof that the system definitively/possibly satisfies the specification. When a property is *definitely satisfied*, the proof specifies to the designer why the search of a definitive/possible counterexample has failed. Instead, whenever a property is *possibly satisfied*, besides providing a possible counterexample, the proof specifies why a definitive counterexample has not been found.

We instantiate THRIVE considering PKSs and LTL formulae. The deductive verification framework presented in Section 3.3 is executed on the top of the three-valued model checker introduced in Section 3.2. The extended version of the paper contains additional details on how these two frameworks are integrated. We discuss the correctness of THRIVE w.r.t the three-valued and the thorough semantics. We also analyze the behavior of the framework for a particular subset of LTL formulae called self-minimizing LTL formulae. For clarification purposes, we summarize in Table 1 the validity of the verification results obtained by the framework.

LTL formulae. *Three-valued LTL semantics.* When the three-valued model checker returns a $\top$ value, no counterexample is returned. By Proposition 1, a

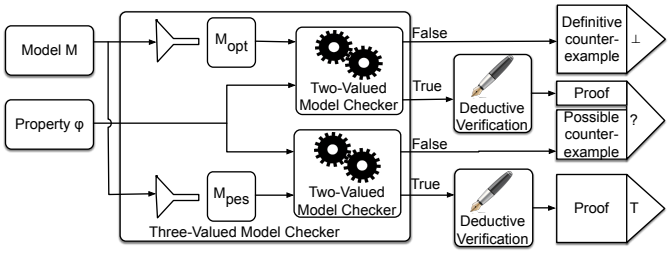

Fig. 6. The proposed three-valued verification framework.

Table 1. LTL and self-minimizing LTL validity of the framework outputs.

		LTL		Self-minimizing LTL	
Result	Type of Output	Three-valued	Thorough	Three-valued	Thorough
$\top$	Verification result	✔	✔	✔	✔
	Proof	✓	✓	✓	✓
?	Possible counterexample	✘	–	✔	✔
	Proof	✗	–	✓	✓
$\bot$	Definitive counterexample	✔	✔	✔	✔

property ϕ evaluated to $\top$ considering the three-valued semantics, is also satisfied considering the thorough semantics. Consequently, this case corresponds to a correct verification and it is marked with a ✔ symbol in Table 1. In these cases, THRIVE uses the deductive verification framework to prove that ϕ is satisfied. Since this property also holds considering the thorough semantics, the proof is *valid* and demonstrates that any completion of M satisfies ϕ. The cases in which THRIVE generates valid proofs are marked with a ✓ symbol.

When the model checker returns a $\bot$ value, the counterexample shows to the designer a behavior that violates the property of interest. By Proposition 1, a property ϕ evaluated to $\bot$ considering the three-valued semantics, is also not satisfied considering the thorough semantics. Thus, the counterexample is a valid counterexample that proves the existence of a completion of M that violates ϕ. For this reason, this case is marked with a ✔ symbol in Table 1.

As specified in Section 3.2, the three-valued model checker returns a ? value more often than expected: there are cases in which a ? is returned but all the completions of the model either satisfy or do not satisfy the property of interest. In this case, the counterexample can be a spurious counterexample, i.e., not valid under the thorough semantics. We indicate the case in which a spurious counterexample can be found with the ✘ symbol. Since when a property ϕ is evaluated to ? there could exist a completion that violates ϕ, the produced proof cannot be considered valid under the thorough semantics. We indicate the case in which it is possible to generate such a proof with the ✗ symbol.

Thorough LTL semantics. As previously discussed, whenever a $\top$ ($\bot$) result is returned by the three-valued model checker, the result is valid also under the thorough semantics. For this reason, THRIVE can be successfully applied in these cases. Moreover, when the three-valued model checker returns a ? value, the result can be not valid. In this case, if a correct result is required, the use of a generalized model checking procedure [5] (not discussed here) becomes necessary. Consequently, a more generic deductive verification framework, developed on top of the generalized model checking procedure, should be used. In Table 1 this case is marked with a – symbol, since currently THRIVE does not support it.

Self-minimizing LTL formulae. Self-minimizing LTL formulae are a subset of LTL formulae that present a really interesting property: the three-valued and the thorough semantics coincide, i.e., if ϕ is self-minimizing, then $[(M, s) \models \phi] = [(M, s) \models \phi]_t$. Therefore, the three-valued model checking framework pre-

sented in Section 3.2 can also be used to verify a property under the thorough semantics. For this reason, in Table 1, the cells associated with the three-valued model checking procedure are marked with ✔. Whenever the three-valued model checker returns $\top$, Proposition 2 certifies that the proof is valid also under the thorough semantics. Moreover, if ? is returned, the proof demonstrates that a definitive counterexample cannot be found. For this reason, in Table 1, the cells associated with the deductive verification procedure are marked with ✓.

5 Preliminary evaluation

This section discusses how THRIVE supports the designer in the development of the system that satisfies the properties of interest. We first consider the example introduced in Section 2 w.r.t. property ϕ_2. Then, properties ϕ_1 and ϕ_3 are also briefly discussed.

Property ϕ_2. The property $\phi_2 = \Box\Diamond green$ specifies that green lights up infinitely often. Even if in state s_1 the green light is on, the property is "only" possibly satisfied. THRIVE returns the possible counterexample $(s_0, s_2)^\omega$, that shows why ϕ_2 is possibly satisfied. Specifically, by looping an infinite number of time on states s_0 and s_2 the green light will not be turned on. This is a possible counterexample since the value of the *green* proposition in s_2 is currently unspecified. Since the property ϕ_2 is possibly satisfied, the search of a definitive counterexample has failed. THRIVE computes a proof that explains the motivation. To search for a counterexample, the intersection automaton M_{opt} presented in Figure 4 is explored by THRIVE during the proof computation. The obtained proof is presented in Table 2, where g stands for *green*. The nodes that have been analyzed in different steps are circled in Figure 4 through different grey frames. (*Step1*). THRIVE analyzes the failed nodes, i.e., the nodes in the set $Fail(I_{opt})$. Since in these nodes the search for a counterexample trivially fails, the formula associated with the corresponding states of $\mathcal{A}_{\neg\phi_2}$ holds. Thus, since the state $\langle s_1, q_1\rangle$ of the intersection automaton is a failed node, the formula $green \vee \bigcirc\Diamond green$ (valid in q_1) is satisfied by the model state s_1. This formula is effectively true in s_1 since the green light is on. By means of a similar reasoning, the algorithm states that the property $green \vee \bigcirc\Diamond green$ also holds in s_2. Indeed, since in the optimistic model the green light is on in s_2, the search of a counterexample has failed in $\langle s_2, q_1\rangle$. (*Step2*). Since all the successors of $\langle s_0, q_1\rangle$ satisfy $green \vee \bigcirc\Diamond green$, it is possible to deduce that this property is also satisfied in s_0. (*Step3*). The induction rule is applied considering the strongly connected component $\{\langle s_0, q_0\rangle, \langle s_1, q_0\rangle, \langle s_2, q_0\rangle\}$. The rule allows to conclude that s_0 satisfies the property $\bigcirc\Box\Diamond green$. (*Step4*). THRIVE applies the conjunction rule to s_0. Since s_0 satisfies both $\bigcirc\Box\Diamond green$ and $green \vee \bigcirc\Diamond green$, it is possibly to deduce that s_0 satisfies the property ϕ_2. This provides an interesting insight to the designer: if she/he does her/his "best" to satisfy the property (she/he turns the green light on in s_2) the property becomes satisfied. The proof clearly states why.

Table 2. Proof that ϕ_2 is not violated.

Step 1	*Step 2*	*Step 3*	*Step 4*
Fail	**Successors**	**Induction**	**Conjunction**
$\langle s_2, q_1\rangle$, $\langle s_1, q_1\rangle$	$\langle s_0, q_1\rangle$	$\mathcal{X} = \{\langle s_0, q_0\rangle, \langle s_1, q_0\rangle, \langle s_2, q_0\rangle\}$ $Exit(\mathcal{X}) = \{\langle s_0, q_1\rangle, \langle s_1, q_1\rangle, \langle s_2, q_1\rangle\}$	The initial node s_0
$s_1 \in Fail(I_{opt})$ $s_2 \in Fail(I_{opt})$ $\hline$ $s_1 \models g \vee \bigcirc\Diamond g$ $s_2 \models g \vee \bigcirc\Diamond g$	$s_0 \rightarrow \{s_1, s_2\}$ $s_1 \models g \vee \bigcirc\Diamond g$ $s_2 \models g \vee \bigcirc\Diamond g$ $\hline$ $s_0 \models g \vee \bigcirc\Diamond g$	$s_0 \models g \vee \bigcirc\Diamond g$ $s_1 \models g \vee \bigcirc\Diamond g$ $s_2 \models g \vee \bigcirc\Diamond g$ $s_0 \rightarrow \{s_1, s_2\}$ $s_1 \rightarrow \{s_0\}$ $s_2 \rightarrow \{s_0\}$ $\hline$ $s_0 \models \bigcirc\Box\Diamond g$ $s_1 \models \bigcirc\Box\Diamond g$ $s_2 \models \bigcirc\Box\Diamond g$	$s_0 \models \bigcirc\Box\Diamond g$ $s_0 \models g \vee \bigcirc\Diamond g$ $\bigcirc\Box\Diamond g \wedge (g \vee \bigcirc\Diamond g) \rightarrow \phi_2$ $\hline$ $s_0 \models \phi_2$

Property ϕ_1. The property $\phi_1 = \Box\Diamond red$ specifies that red lights up infinitely often. The designer wants to know whether the model of Figure 1 satisfies ϕ_1. Intuitively, it is sufficient to observe that the proposition *red* is evaluated with $\top$ in the state s_0 and that the system always returns to this state (after visiting either s_1 or s_2). THRIVE returns $\top$ as its verification result and produces a proof that highlights how and why a definite counterexample is not found in the graph. First, it identifies the nodes $\langle s_0, q_1\rangle$ and $\langle s_2, q_1\rangle$ as failed. The conclusions found on these nodes are propagated to the node $\langle s_1, q_1\rangle$. All the successors of the SCC formed by the intersection states related to the property state q_0 are analyzed. Therefore conclusions are drawn also on this SCC. The proof is not reported for brevity and it follows the same pattern as the one in Table 2.

Property ϕ_3. $\Box(red \rightarrow \Box green)$ cannot be satisfied by the chosen model. Therefore THRIVE returns a definite counterexample, e.g. $(s_0, s_1)^\omega$.

6 Discussion

This section elaborates on the applicability of THRIVE.

Three-valued vs thorough semantics. As previously discussed, THRIVE does not produce a useful result when a thorough semantics of LTL formulae is considered and a ? value is returned. Indeed, the property could be $\top$, ? or $\bot$ w.r.t. the thorough semantics. The generalized model checking algorithm [5](which levies a performance penalty) could be used to discriminate between these cases. In [16], the authors analyze how often this additional check really helps. They show that whenever the model is built using predicate abstraction [13], the thorough check does not provide additional precision[1]. It is also argued that in many

[1] The results of [16] are presented for CTL properties; however the authors specify that the results naturally extend to LTL.

practically interesting cases, the thorough semantics is not more precise than the three-valued one. For these reasons, THRIVE can be usefully applied in most of the real world applications.

Temporal patterns of self-minimization. THRIVE always produces a correct result when the LTL formula is self-minimizing. In [10] the authors propose a first grammar for this LTL subset. The grammar can be used by the designer to generate formulae that are (by construction) self-minimizing, or to check whether a specific formula is self-minimizing, i.e., if the formula is compliant with the grammar, then it is self-minimizing. The authors also argue that the set of self-minimizing LTL formulae contains most of the properties of practical interest, such as absence, universality, existence, response and response chain. The grammar presented in [10] is improved in [1], where the authors consider popular syntactic specification patterns, documented at a community-led pattern repository, and check whether the formulae that are compliant with these patterns are self-minimizing. It is showed that many such patterns are self-minimizing and that, the ones that are not, can be transformed with linear blowup into a self-minimizing LTL formula. Thus, in most cases, the designer will consider a formula that is self-minimizing. A syntactic check can be used to prove self-minimization before running THRIVE.

Checking whether an LTL formula is self-minimizing. Checking whether an LTL formula is self-minimizing (if it is not compliant with one of the syntactic patterns) is expensive, since it requires to compute an automaton that is exponential in $|\phi|$ [10]. However, if ϕ satisfies some constraints (sufficient conditions) then it is self-minimizing. For example, if it is in its negation normal form and no proposition occurs in mixed polarity, then ϕ is self-minimizing. These checks can be implemented in THRIVE. Note that, some LTL formulae can be transformed into their equivalent self-minimizing version, but not all the LTL formulae have a semantic minimization in LTL [10].

Scalability. Three-valued model checking is as expensive as classical model checking [4], which is commonly used to analyze real world problems [26]. Deductive verification has been employed successfully in the verification of digital hardware and software systems [24]. However, there are inherent limits to the efficiency with which expressive general-purpose logics can be fully mechanized. Two approaches have been proposed in literature to overcome this limitation: 1. using interactive deductive verification tools so that correctness proofs can be developed through a combination of user guidance and limited forms of automated deduction; 2. considering useful fragments of logic that can be mechanized very effectively. Since THRIVE simply combines multi-valued model checking and theorem proving, its scalability improves as the performance of the employed model checking and deductive verification frameworks enhances.

7 Related Work

Three-valued [18, 11, 4, 5, 3, 12] and multi-valued [14, 6] model checking techniques have been used to verify models that are incomplete, i.e., in which some

information is missing. Two types of semantics are usually considered in these works: compositional (three-valued) and/or non-compositional (thorough). The compositional semantics exploits the Kleene algebraic structure between the values $\{false, \bot, true\}$. The non-compositional semantics is based on the completeness preorder. These two semantics have been analyzed in many works, such as [16, 10]. Depending on the modeling formalism, different model checking techniques have been developed. For example, several works proposed in literature consider Partial Kripke Structures (e.g., [4, 5, 12, 14, 6]), others Modal Transition Systems (e.g., [18, 11]). However, to the best of our knowledge, none of these approaches has been combined with deductive verification.

Deductive verification includes a set of techniques that establish the validity of the formula (for more information see [19]). Some works have used deductive verification as a means to enrich the information returned by model checking [15, 7]. In other techniques [20, 25, 24], the idea is to exploit the structure of the state space generated by the model checker to explain why a property holds. To the best of our knowledge, none of these approaches has been applied in a multi-valued context.

8 Conclusions

We have proposed THRIVE. This framework enhances a three-valued model checker with a deductive verification engine. Whenever the property of interest is definitely satisfied, or possibly satisfied, THRIVE provides the designer with additional information regarding why a certain result is returned by the model checker. The proof gives intuition on what is working correctly in the current design and insights for the next development rounds. The presented framework can be implemented on top of existing model checking tools based on automata theory.

THRIVE has been evaluated considering a simple semaphore example, which showed the usefulness of the approach. We proved that the result generated by THRIVE is always valid when self-minimizing LTL formulae are considered. We discussed that self-minimizing formulae can be constructed following specific patterns and that checking whether an LTL formula is self-minimizing can be done using appropriate procedures. We also argued that most of the properties of practical interest are already self-minimizing and that some LTL formulae can be transformed into a self-minimizing version. We showed that when non self-minimizing LTL formulae are considered, THRIVE produces a valid result in most of the cases: when a $\top$ or $\bot$ value is returned, and also when a ? value is returned and the formula is possibly satisfied under the thorough semantics. Finally, we pointed out that THRIVE improves as model checking and deductive verification frameworks enhance.

As future work, we aim to implement THRIVE by integrating existing model checking and theorem proving frameworks. Additionally, we will prove the usefulness of THRIVE over a real world example.

References

1. A. Antonik and M. Huth. Efficient patterns for model checking partial state spaces in CTL ∩ LTL. *Electronic Notes in Theoretical Computer Science*, 158:41–57, 2006.
2. C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.
3. N. Beneš, J. Křetínskỳ, K. G. Larsen, and J. Srba. Checking thorough refinement on modal transition systems is exptime-complete. In *Theoretical Aspects of Computing*, pages 112–126. Springer, 2009.
4. G. Bruns and P. Godefroid. Model checking partial state spaces with 3-valued temporal logics. In *Computer Aided Verification*, pages 274–287. Springer, 1999.
5. G. Bruns and P. Godefroid. Generalized model checking: Reasoning about partial state spaces. In *International Conference on Concurrency Theory*, pages 168–182. Springer, 2000.
6. G. Bruns and P. Godefroid. Model checking with multi-valued logics. In *Automata, Languages and Programming*, pages 281–293. Springer, 2004.
7. E. M. Clarke, O. Grumberg, K. L. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Design Automation Conference*. ACM, 1995.
8. E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.
9. R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification, Testing and Verification*, pages 3–18. Springer, 1996.
10. P. Godefroid and M. Huth. Model checking vs. generalized model checking: semantic minimizations for temporal logics. In *Logic in Computer Science*, pages 158–167. IEEE Computer Society, 2005.
11. P. Godefroid, M. Huth, and R. Jagadeesan. Abstraction-based model checking using modal transition systems. In *Concurrency Theory*, pages 426–440. Springer, 2001.
12. P. Godefroid and N. Piterman. LTL generalized model checking revisited. *International journal on software tools for technology transfer*, 13(6):571–584, 2011.
13. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Computer Aided Verification*, pages 72–83. Springer, 1997.
14. A. Gurfinkel and M. Chechik. Multi-valued model checking via classical model checking. In *Concurrency Theory*, pages 266–280. Springer, 2003.
15. A. Gurfinkel and M. Chechik. Proof-like counter-examples. In *In Tools and Algorithms for the Construction and Analysis of Systems*, pages 160–175. Springer, 2003.
16. A. Gurfinkel and M. Chechik. How thorough is thorough enough? In *Correct Hardware Design and Verification Methods*, pages 65–80. Springer, 2005.
17. S. A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16(1963):83–94, 1963.
18. K. G. Larsen and B. Thomsen. A modal process logic. In *Logic in Computer Science*, pages 203–210. IEEE, 1988.
19. Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems: Specification*. Springer, 1992.
20. K. S. Namjoshi. Certifying model checkers. In *Computer Aided Verification*, pages 2–13. Springer, 2001.
21. D. Peled, A. Pnueli, and L. Zuck. From falsification to verification. In *Foundations of Software Technology and Theoretical Computer Science*, pages 292–304. Springer, 2001.

22. D. Peled and L. Zuck. From model checking to a temporal proof. In *International SPIN workshop on Model checking of software*, pages 1–14. Springer, 2001.
23. A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science*, pages 46–57. IEEE, 1977.
24. S. Rajan, N. Shankar, and M. K. Srivas. An integration of model checking with automated proof checking. In *Computer Aided Verification*, pages 84–97. Springer, 1995.
25. L. Tan and R. Cleaveland. Evidence-based model checking. In *Computer Aided Verification*, pages 455–470. Springer, 2002.
26. J. Woodcock, P. G. Larsen, J. Bicarregui, and J. S. Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys*, 41(4), 2009.