From model checking to a temporal proof for partial models

Anna Bernasconi[1], Claudio Menghi[1], Paola Spoletini[2], Lenore D. Zuck[3], and Carlo Ghezzi[1]

Politecnico di Milano, DEIB - DEEPSE group
{anna.bernasconi,claudio.menghi,carlo.ghezzi}@polimi.it,
pspoleti@kennesaw.edu
[2] Kennesaw State University,
lenore@cs.uic.edu
[3] University of Illinois at Chicago

Abstract. Three-valued model checking has been proposed to support verification when some portions of the model are unspecified. Given a formal property, the model checker returns *true* if the property is satisfied, *false* and a violating behavior if it is not, *maybe* and a possibly violating behavior if it is *possibly satisfied*, i.e., its satisfaction may depend on how the unspecified parts are refined. Model checking, however, does not explain the reasons *why* a property holds, or possibly holds. Theorem proving can instead do it by providing a formal proof that explains why a property holds, or possibly holds in a system. Integration of theorem proving with model checking has only been studied for classical two-valued logic – hence, for fully specified models. This paper proposes a unified approach that enriches three-valued model checking with theorem proving to generate proofs which explain why *true* and *maybe* results are returned.

1 Introduction

Multi-valued model checking techniques, such as [4, 5, 14, 6], have been proposed to support the verification of models that are *partial*, i.e., their state space is not fully specified. Three-valued model checking is a multi-valued model checking technique that extends classical two-valued model checking by possibly returning an additional *maybe* value. More precisely, it returns *true* if the property definitely holds, *false* if it definitely does not hold, *maybe* otherwise.

In the classical context of two-valued model checking, although a sample violating behavior (a counterexample) is normally returned when the property is violated, no equally useful insight is provided if the property holds. In practice, it would be useful to receive a formal explanation of the reason *why* the system satisfies the property. To achieve this goal, the model checking framework can be equipped with a theorem prover that formally justifies why model checking has failed in the search of a counterexample. Theorem proving algorithms have been developed for fully specified models [19, 20], but no known similar approach deals with partial models.

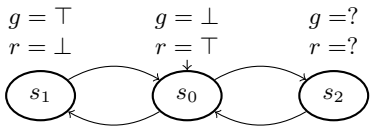

Fig. 1. System model M

The ability to deal with partial models has a strong practical motivation. Software development often proceeds in an iterative and incremental fashion. Designers 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 result of verification is *true* or *maybe*, the proof can guide the designer throughout the refinement process, and confirm the correctness of the design choices already performed. In some cases, the proof may even implicitly suggest that actually the property does not capture the intended correctness condition, and it should be modified. For this reason, the integration of theorem proving techniques and multi-valued model checking can guide the designer towards the development of a correct model.

This paper proposes THRIVE, a THRee valued Integrated Verification framEwork for partial models. THRIVE enriches model checking for partial models with theorem proving. Theorem proving is used when a *true* or a *maybe* value is returned by the model checker to justify why the verified system *definitely* or *possibly* satisfies the property of interest. In addition to the general framework, we present a specific instance of THRIVE useful for applications, which considers models described as Partial Kripke Structures (PKSs) [4] and properties expressed as Linear Temporal Logic (LTL) [21] formulae. The instance is based on the three-valued LTL semantics [4]. To successfully integrate model checking and theorem proving we customize the theorem proving framework (based on deductive verification) proposed in [20] to support PKSs and LTL formulae.

We consider the applicability of THRIVE w.r.t. *three-valued* [4] and *thorough* [5] LTL semantics. We also discuss its applicability in the case of *self-minimizing* [10] LTL formulae, which are known to represent a practically relevant subset of LTL formulae [2]. We evaluate the benefits of the framework on an example by simulating the design of a medical software critical component [3]. A discussion on the use of THRIVE in real world scenarios concludes the evaluation.

Running example. *We consider a simple grade crossing semaphore. We assume that the designer has identified three simple properties: (1) Red lights up infinitely often – formalized as* $\phi_1 = \Box\Diamond red$. *(2) Green lights up infinitely often – formalized as* $\phi_2 = \Box\Diamond green$. *(3) When the light is red, it will always be green – formalized as* $\phi_3 = \Box(red \rightarrow \Box green)$. *Note that* ϕ_3 *is deliberately wrong and will be used later to discuss the application of THRIVE.*
Starting from this specification, a designer might initially propose the partially 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. 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 g and r to either $\top$ or $\bot$ in s_2.

Related work. Three-valued [16, 11, 4, 5, 12] and multi-valued [14, 6] model checking has been proposed to support verification of partial models. Different model checking techniques have been developed depending on the modeling formalisms. For example, several papers focus on Partial Kripke Structures (e.g., [4, 5, 12, 14, 6]), others on Modal Transition Systems (e.g., [16, 11]). However, to the best of our knowledge, none of these techniques has been combined with theorem proving.

Theorem proving applies a set of techniques to try to establish the validity of a given formula (see [17]). Some of these techniques (e.g., [19, 20, 18, 23, 22]) exploit the state space generated by the model checker to explain why a property holds. However, to the best of our knowledge, none of these approaches has been applied in a multi-valued context.

Organization. Section 2 presents background notations and algorithms. Section 3 describes THRIVE. Section 4 presents an instance of THRIVE, that considers PKSs and LTL formulae. Section 5 evaluates the approach on an example. Section 6 discusses the applicability of THRIVE in real world cases. Finally, Section 7 concludes the paper.

2 Background

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

Checking partial models. *Partial Kripke Structures* [4] (PKSs) extend KSs by allowing a proposition in a given state to be labelled with ? to represent an unknown value. 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 model of the grade crossing semaphore presented in Figure 1 is an example of a PKS.

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

Two kinds of LTL semantics (three valued and thorough) exists for PKSs.

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. According to three-valued 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, we inductively define that π satisfies ϕ in the model M as follows:

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

where the notation π^i indicates the sub-path $s_i, s_{i+1} \ldots$ of π.

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

Given a PKS $M = \langle S, R, S_0, AP, L\rangle$, satisfaction of formula ϕ in a state s is defined as $[(M,s) \models \phi] = \min(\{[(M,\pi) \models \phi] \mid \pi^0 = s\})$. 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.

Three-valued semantics does not behave always in accordance with the natural intuition [5]: there are cases in which ϕ possibly holds for a PKS but all its completions actually satisfy (or do not satisfy) ϕ. For this reason, an alternative semantics, called *thorough LTL semantics* [5] has been proposed. According to it, a formula is possibly satisfied only if there exist two completions $M_1 \in \mathcal{C}(M)$ and $M_2 \in \mathcal{C}(M)$, such that ϕ is definitely satisfied in one and violated in the other. Thorough semantics defines satisfaction of an LTL formula ϕ by a PKS M ($[M \models \phi]_t$) as follows:

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

Given a PKS and an LTL formula ϕ, it has been proved [12] that (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.

There exists a subset of LTL formulae, known in the literature as *self-minimizing* [10], such that the two semantics coincide. Formally, given a model M and a self-minimizing LTL property ϕ, then $[M \models \phi] = [M \models \phi]_t$. It has been observed that most practically useful LTL formulae belong to this subset [10].

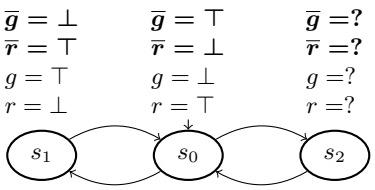

Fig. 2. The PKS of the crossing semaphore.

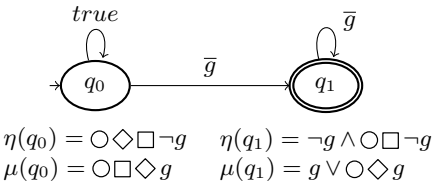

Fig. 3. The BA associated with ϕ_2.

We present a *model checking* algorithm for PKSs and LTL formulae based on three-valued semantics. This procedure considers a version of M, called *complement-closed* [5], in which for every proposition $p \in AP$, there exists a new proposition $\overline{p}$, called complement-closed proposition, such that $L(s,p) = \text{comp}(L(s,\overline{p}))$, for all $s \in S$. For example, the complement-closed version of the PKS of the semaphore example is presented in Figure 2.

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 proposition of the complement-closure of M with value ?. 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 case, 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. 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 cases. Then, $[(M,s) \models \phi]$ has been defined in [5][4] as:

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

This technique exploits two runs of the classical two-valued model checking performed on a pessimistic and an optimistic completion of M.

Deductive verification. Hereafter we briefly recall background information on how to integrate a theorem prover with a model checker. Given a complete

[4] In [5] the procedure is presented for Positive Propositional Modal Logic but has been proved to be valid also for LTL (see [5, 10, 12]).

KS M and an LTL property ϕ that is satisfied by M, the deductive verification framework produces a proof which explains why $M \models \phi$ [20] considering the intersection $\mathcal{G} = M \otimes \mathcal{A}_{\neg\phi}$. The approach is based on three considerations. (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 $\mathcal{G}$, $\sigma_i \models \eta(q_i)$. The formula $\eta(q)$ 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 have been found. Thus, the negation $\mu(q)$ of $\eta(q)$ holds in s; (3) Each state $\langle s, q\rangle$ 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 states, called *failed state*, 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 enriches the intersection $M \otimes \mathcal{A}_{\neg\phi}$ by considering also failed states as part of it. Since in each failed state $\langle t, p\rangle$ the search of a counterexample has failed, we can write the failure axiom $t \models \mu(p)$. A set of deductive rules is applied to produce the proof. (1) *Successors rule.* Given a state $\langle s, q\rangle$ of the intersection, 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 does not violate 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 states $\langle s_i, q_j\rangle$ that do not belong to $\mathcal{X}$ and have an incoming transition from a source state in $\mathcal{X}$. If every state $\langle s_i, q_j\rangle \in Exit(\mathcal{X})$ is such that $s_i \models \mu(q_j)$, we can conclude that, for every state $\langle s, q\rangle \in \mathcal{X}$, $s \models \mu(q)$ holds. Intuitively, since all the "successors" of $\mathcal{X}$ (the states 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), it must be that each state s satisfies the corresponding property $\mu(q)$. (3) *Conjunction rule.* It connects conclusions made on a given state making temporal logic interferences. The formulae computed for a given state 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 $\mathcal{X}'$.

3 THRIVE

An overview of THRIVE is presented in Figure 4. THRIVE takes as inputs a partial model M and a property ϕ and produces one of the outputs shown by

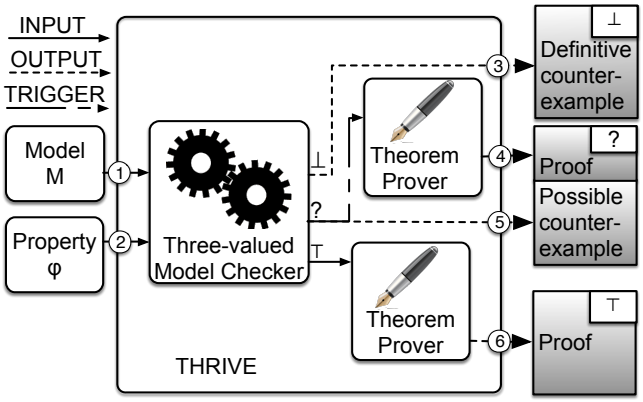

Fig. 4. THRIVE.

the grey filled shapes. The outputs are generated by integrating a model checker for partial models and a theorem prover.

The *model checker for partial models* verifies 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* (③), there exist some behaviors which definitively violate the property of interest and do not depend on the unspecified parts of the model. The model checker returns one such behavior, i.e., a definitive counterexample. Whenever a property is *definitely satisfied* (⑥), its satisfaction does not depend on the unspecified parts, i.e., on how the incomplete parts are later refined. Finally, if the property is *possibly satisfied* (④, ⑤), the model checker returns a possible counterexample, i.e., a possible violating behavior that the model can exhibit.

The *theorem proving* framework is executed when a $\top$ or ? value is returned by the model checker and is used to compute a proof which specifies why the property ϕ is definitely (possibly) satisfied by M. When a property is *definitely satisfied*, THRIVE returns a proof that specifies why the search of a definitive and a possible counterexample has failed. Instead, whenever a property is *possibly satisfied*, besides providing a possible counterexample, THRIVE returns a proof that specifies why a definitive counterexample has not been found.

4 Using THRIVE with PKS and LTL

This section describes the instance of THRIVE proposed in this paper, using PKSs and and LTL. We first show how we modified the theorem prover framework presented in Section 2 to support PKSs and how it is integrated with the three-valued model checker. We further analyze the case of thorough semantics, which is more appealing in practice, and discuss to what extent and how the framework can be used in such a case.

4.1 Adapting the theorem prover.

The deductive verification framework presented in [20] exploits the intersection between a state labeled transition systems and a Generalized Büchi Automaton (GBA [9]) to generate the proof. To enable the algorithm to work on KSs and a BA, it is necessary to specify how to associate LTL formulae with each state of the BA and how to identify failed states of the intersection automaton.

Identification of the formulae that hold in the states of the BA. We assume that the degeneralization procedure [7], that converts the GBA $\mathcal{G}$ into an equivalent BA $\mathcal{A}$, behaves as follows: when a new state q of $\mathcal{A}$ is created from a state q' of $\mathcal{G}$, the formulae $\eta(q')$ and $\mu(q')$ are also associated to q.

Identification of failed states. Following the procedure mentioned in Section 2, the product automaton $M \otimes \mathcal{A}$ between the KS M and the BA $\mathcal{A}$ is modified to also generate *failed states*. Specifically, the intersection is computed using the rules 1 and 2.

$$\frac{s \to t \wedge q \xrightarrow{L(t)} p}{\langle s,q\rangle \to \langle t,p\rangle} \quad (1) \qquad \frac{s \to t \wedge q \xrightarrow{\not L(t)} p}{\langle s,q\rangle \dashrightarrow \langle t,p\rangle} \quad (2)$$

Rule 1 is the classical rule used to compute the intersection automaton. It specifies that the state of the intersection $\langle s,q\rangle$ moves to $\langle t,p\rangle$ only if the transition $q \xrightarrow{L(t)} p$ that moves the BA from q to p has the same label of the state t of M. Rule 2 specifies how to compute failed states. It states that the failed state $\langle t,p\rangle$ is generated in the intersection when a transition that moves the BA $\mathcal{A}$ from q to p is labelled differently with respect to the state t reached by the model $\mathcal{M}$ when the transition $s \to t$ is fired. This is indicated using the notation $q \xrightarrow{\not L(t)} p$. For this reason, the transition $\langle s,q\rangle \dashrightarrow \langle t,p\rangle$ from $\langle s,q\rangle$ to $\langle t,p\rangle$ is dashed. Let us consider the intersection presented in Figure 5 computed from the KS M_{opt} obtained from the PKS in Figure 2 and the BA of Figure 3. The transition $\langle s_0,q_0\rangle$ to $\langle s_1,q_1\rangle$ of the intersection presented in Figure 5 is dashed, since the proposition $\overline{g}$ is false in s_1, while the labeling of the transition from q_0 to q_1 requires $\overline{g}$ to be true for the transition to be performed.

The set $\mathcal{F}(M \otimes \mathcal{A})$ of the *failed states* contains the states $\langle t,p\rangle$ obtained by applying rule 2. Note that, as stated in Section 2, each failed state $\langle s,q\rangle$ is such that $s \models \mu(q)$. For example, the state $\langle s_1,q_1\rangle$ of the intersection presented in Figure 5 is a failed state. Indeed, s_1 satisfies the property $\mu(q_1) = g \vee \bigcirc\Diamond g$ associated with the state q_1.

Theorem 1. *The deductive verification procedure is correct.*

Proof. We show that the states identified as *failed* correspond to the ones that would be identified using [20]. In [20], a state $\langle t,p\rangle$ is failed if the propositional assignment of t does not satisfy the conditions specified in the state p. It is well known [9, 7], that a GBA $\mathcal{G}$ associated with ϕ is such that (1) all the transitions $(q,\alpha,p) \in \Delta$ that reach a state p of the GBA have the same label α and that

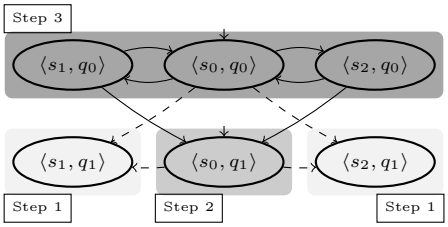

Fig. 5. Product $I_{opt} = M_{opt} \otimes \mathcal{A}_{\neg\phi_2}$

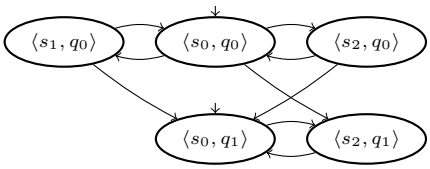

Fig. 6. Product $I_{pes} = M_{pes} \otimes \mathcal{A}_{\neg\phi_2}$

(2) a transition $(q, \alpha, p) \in \Delta$ is in the GBA if and only if α satisfies the conjunction of the negated and non negated propositions that hold in the state p. By construction, the latter of these properties also holds in the BA obtained from the GBA by applying the degeneralization procedure [7]. Thus, since all the transitions that reach p are labelled with α, a transition $\langle s, q\rangle \dashrightarrow' \langle t, p\rangle$ is added to the product automaton if and only if the propositional assignment of t does not satisfy the propositional assignment specified in the state p. Furthermore, BAs acceptance condition is a special case of fairness condition used in [20]. Thus, the proposed deductive verification technique is a special case of [20], with regard to acceptance. □

4.2 Integrating the model checker and the theorem prover

Figure 7 presents an instance of THRIVE obtained as an integration of a model checker for PKSs and LTL based on three-valued semantics and the theorem prover presented in Section 4.1. The circled numbers in Figure 7 indicate how this specific instance is plugged into THRIVE in Figure 4.

The three-valued model checker presented in Section 2 is used by THRIVE to check the satisfaction of the property of interest. Specifically, it runs twice a classical two-valued model checker, considering first the optimistic approximation M_{opt}, then the pessimistic approximation M_{pes} of the PKS M. When M_{opt} is evaluated, if a counterexample is found, this is returned as output of THRIVE. Otherwise, THRIVE verifies M_{pes}. If the property is satisfied, it means that no violating nor possibly violating behaviors have been identified. Thus, THRIVE executes the theorem prover that produces a proof that explains why no counterexample has been found in the pessimistic approximation. Otherwise, the property is possibly satisfied. In this case, THRIVE returns the possible counterexample and runs the theorem prover on M_{opt} to compute a proof that specifies why a definitive counterexample has not be found.

Example *Consider properties ϕ_1, ϕ_2 and ϕ_3 of the crossing semaphore example. They are satisfied, possibly satisfied and not satisfied, respectively, by the model M of Figure 2.*

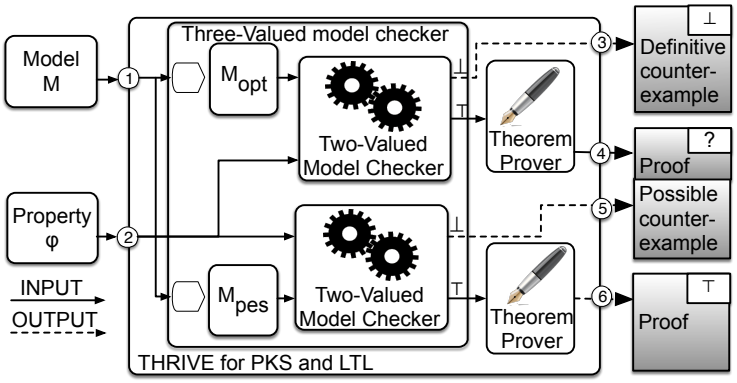

Fig. 7. THRIVE for PKS and LTL.

Property ϕ_2*. The intersections between the optimistic and pessimistic approximation of the model* M *and the BA automaton* $\mathcal{A}_{\neg\phi_2}$ *are presented in Figures 5 and 6. THRIVE explores* I_{pes} *and returns the possible counterexample* $(s_0, s_2)^\omega$*. Specifically, by looping an infinite number of times on states* s_0 *and* s_2 *the green light is never turned on. Since the property* ϕ_2 *is possibly satisfied, the search of a definitive counterexample in the intersection automaton* I_{opt} *(presented in Figure 5) fails. THRIVE uses the intersection automaton* I_{opt} *to compute a proof that explains the motivation. The obtained proof is presented in Table 1. The states that are analyzed in different steps are circled in Figure 5 through different grey frames. (*Step1*). THRIVE analyzes the failed states. Given a failed state* $\langle s, q\rangle$*, since in this state the search for a counterexample fails, the formula associated with the state* q *of* $\mathcal{A}_{\neg\phi_2}$ *holds in* s*. For example, since the state* $\langle s_1, q_1\rangle$ *of* I_{opt} *is a failed state, 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. (*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 turns the green light on in* s_2 *the property becomes satisfied. The proof clearly states why.*

Property ϕ_3*. THRIVE returns the counterexample* $(s_0, s_1)^\omega$*. The counterexample specifies that by looping an infinite number of times on states* s_0 *and* s_1 *the green light is not permanently on after the red.*

Property ϕ_1*. THRIVE produces a proof that highlights how and why a definite counterexample is not found in the graph. First, it identifies the states* $\langle s_0, q_1\rangle$ *and* $\langle s_2, q_1\rangle$ *as failed. The conclusions found on these states are propagated to*

Table 1. 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 state s_0
$\langle s_1, q_1\rangle \in \mathcal{F}(I_{opt})$ $\langle s_2, q_1\rangle \in \mathcal{F}(I_{opt})$ / $s_1 \models g \vee \bigcirc\Diamond g$ $s_2 \models g \vee \bigcirc\Diamond g$	$s_0 \rightarrow \{s_1, s_2\}$ $s_1 \models g \vee \bigcirc\Diamond g$ $s_2 \models g \vee \bigcirc\Diamond g$ / $s_0 \models g \vee \bigcirc\Diamond g$	$s_0 \models g \vee \bigcirc\Diamond g$ $s_1 \models g \vee \bigcirc\Diamond g$ $s_2 \models g \vee \bigcirc\Diamond g$ $s_0 \rightarrow \{s_1, s_2\}$ $s_1 \rightarrow \{s_0\}$ $s_2 \rightarrow \{s_0\}$ / $s_0 \models \bigcirc\Box\Diamond g$ $s_1 \models \bigcirc\Box\Diamond g$ $s_2 \models \bigcirc\Box\Diamond g$	$s_0 \models \bigcirc\Box\Diamond g$ $s_0 \models g \vee \bigcirc\Diamond g$ $\bigcirc\Box\Diamond g \wedge (g \vee \bigcirc\Diamond g) \rightarrow \phi_2$ / $s_0 \models \phi_2$

the state $\langle s_1, q_1\rangle$. All the successors of the SCC formed by the intersection states related to the property state q_0 are analyzed. Finally, conclusions are drawn also on this SCC. The proof is omitted for space reasons.

4.3 Thorough semantics and THRIVE

As stated in Section 2, three-valued semantics does not always behave in accordance with the natural intuition [5]. When ϕ possibly holds in M, it is desirable that there exist two completions M' and M'' of M such that M' satisfies ϕ and M'' violates ϕ. This property is not ensured by the three-valued semantics, and is the motivation that leads to introduce thorough LTL semantics. Hereafter, we discuss how the adoption of thorough semantics would affect the use of the THRIVE framework.

Given a PKS M and a property ϕ, THRIVE can produce the following outcomes:

Property is satisfied. In this case, THRIVE works correctly. A property ϕ that evaluates to $\top$ under three-valued semantics is also satisfied under thorough semantics. Thus, the verification result is correct. Also the proof is correct since it shows that any completion of M satisfies ϕ.

Property is not satisfied. In this case, THRIVE works correctly. When the model checker returns a $\perp$ value, the counterexample shows a behavior that violates ϕ. A property ϕ that is not satisfied considering the three-valued semantics, is also not satisfied considering the thorough semantics. Thus, the counterexample is a correct counterexample that proves the existence of a completion of M that violates ϕ.

Property is possibly satisfied. This case is not handled by THRIVE correctly for all LTL properties. When the three-valued model checker returns ? the property is possibly satisfied considering the three-valued semantics but no

conclusion can be drawn based on thorough semantics. Indeed, there are cases in which a ? is returned, but all the completions of the model either satisfy or do not satisfy ϕ. The computed counterexample and proof can be spurious under the thorough semantics.

Example. *The results obtained for ϕ_1 and ϕ_3 of the crossing semaphore example are correct both considering the three-valued and the thorough semantics. Since ϕ_1 is satisfied, the proof is a correct proof that justifies why all the completions of the model presented in Figure 1 satisfy ϕ_1. The counterexample returned for ϕ_3 is correct, i.e., all the completions of the model presented in Figure 1 exhibit the behavior returned as a counterexample.*

Self-minimizing LTL formulae. Self-minimizing LTL formulae are a subset of LTL formulae that present an interesting property: three-valued and thorough semantics are equivalent, i.e., if ϕ is self-minimizing, then $[(M, s) \models \phi] = [(M, s) \models \phi]_t$. Therefore, the three-valued model checking framework presented in Section 2 produces a result that is correct also considering the thorough semantics. For this reason, whenever the three-valued model checker returns ?, the proof and the possible counterexample produced by THRIVE are also correct under the thorough semantics. In [10], the authors propose a first grammar for this LTL subset. The grammar does not capture entirely this set. However, it can be used to generate formulae that are self-minimizing by construction, or to check whether a formula is self-minimizing (sufficient condition). Furthermore, the authors argue that the set of self-minimizing LTL formulae contains most property patterns of practical interest, such as absence, universality, existence, response and response chain [8].

For these reasons it is possible in practice to use the version of THRIVE of Figure 7 also under the thorough semantics interpretation.

Example *Property ϕ_2 is a special instance of LTL response pattern which, according to [10], is self-minimizing. Thus, the possible counterexample and the proof returned by THRIVE are correct.*

5 Preliminary evaluation

This section tries to answer the following research question: *how effective is THRIVE w.r.t. incremental development?*

To provide an initial answer, we simulated the design of a critical software system[5]. The system, described in [3], is used by optometrists and ophtalmologists to test visual problems and certify a certain level of stereoacuity. The test requires patients to pass levels with increasing difficulties, in which they have to recognize images. Each time the patient is able to recognize an image the system shifts to a higher level and a more difficult image is shown. When the patient fails, the level is decreased. The test ends in one of these cases: 1. when the patient fails the image recognition and she/he did not pass an easier level; 2. when the top level is reached; 3. if the doctor interrupts the test.

[5] The model and the obtained results can be found at `https://goo.gl/U680Yn`

Experimental setup. We modeled this system in [3] as a PKS. For simplicity we considered only two levels. We used the atomic propositions *fl*, *sl*, *test*, *edb*, *cert* and *uncert* to specify that the patient is in the first or in the second level of the test, the test is under execution, a mistake has been made by the patient, the patient has been certified and the patient is not certified, respectively. If at some point the doctor quits the test, the patient is not certified. If the patient fails the first level, the patient is not certified. If he/she passes the first level, the second level is entered. If the patient also passes the second level he/she is certified at the second level. Otherwise, we assume that the designer is uncertain on the level in which the component should certify/not-certify the patient (this is formalized by setting $fl =?$, $sl =?$).

We designed a set of properties that the system has to satisfy:

- $\psi_1 = (\neg cert)\,\mathcal{W}(\neg sl)$ states that a patient is not at the second level before he/she is certified (see [1]);
- $\psi_2 = \Box(test \rightarrow \Diamond(cert \vee uncert))$ specifies that every test must be followed by a certification or a non-certification;
- $\psi_3 = \Box(edb \rightarrow \Diamond(cert \vee fl))$ states that if an error has been made by the patient (edb), she/he cannot be uncertified and be at the second level ($\neg fl$). Indeed, a mistake prevents a patient from increasing the assessed level.

Note that these properties are obtained from well-known property patterns [8].

Results. *Property* ψ_1. THRIVE returns the value $\bot$ and returns a definitive counterexample showing that there exists a case in which a patient is assessed at the second level but has not been certified yet. Indeed, the property is wrong, the desired property should have been expressed as $\neg(cert \wedge fl)\,\mathcal{W}(\neg sl)$, meaning that a patient is not at the second level before he/she is certified at the first level.

Property ψ_2. THRIVE returns the value $\top$, since the property of interest is satisfied. The proof shows that a *test* is always followed by a *cert* or *uncert*.

Property ψ_3. THRIVE returns the value ? and a possible counterexample obtained by assigning $\bot$ to the proposition *fl*. THRIVE considers the optimistic approximation to produce a proof that no definitive counterexample can be found. The obtained proof is correct since a simple grammar check shows that ψ_3 is self-minimizing. The proof shows why, by assigning $\top$ to the unknown proposition *fl*, the property of interest is satisfied.

The feedback produced by THRIVE for properties ψ_1, ψ_2 and ψ_3 successfully helps in understanding whether a property of interest is satisfied, possibly satisfied or violated. When the property is satisfied/possibly satisfied, understanding the reason why this is true supports self-confidence.

6 Using THRIVE in real cases

This section elaborates on the applicability of THRIVE in real cases.

Three-valued vs thorough semantics. The generalized model checking algorithm [5] (which levies a performance penalty) could be used to check a property under the thorough semantics. In [15], the authors analyze how the generalized

model checking really helps. Whenever the model is built using predicate abstraction [13], the thorough check does not provide additional precision. It is also argued that in many practically interesting cases, the thorough semantics is not more precise than the three-valued one. For these reasons, THRIVE can be correctly applied in most of the real world cases.

Temporal patterns of self-minimization. In [2], the authors consider popular syntactic specification patterns, documented at a community-led pattern repository, and check whether formulae compliant with these patterns are self-minimizing. They show that many such patterns are self-minimizing and the ones that are not can be transformed with linear blowup into a self-minimizing LTL formula. Thus, in most practical 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 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.

Scalability. Three-valued model checking is as expensive as classical model checking [4], which is commonly used to analyze real world problems [24]. Deductive verification has been employed successfully in the verification of digital hardware and software systems [22]. 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 Conclusions

We have proposed THRIVE, a framework that integrates a three-valued model checker with a theorem prover. Whenever the property of interest is definitely satisfied, or possibly satisfied, THRIVE provides 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. THRIVE has been evaluated considering a safety critical example [3], which showed the effectiveness of the approach. We also showed the applicability of the approach in real world cases. As future work, we aim to implement THRIVE by integrating existing model checkers and theorem provers.

References

1. H. Alavi, G. Avrunin, J. Corbett, L. Dillon, M. Dwyer, and C. Pasareanu. Spec patterns, 2017. Available at `http://patterns.projects.cs.ksu.edu/documentation/patterns/ltl.shtml`.
2. 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.

3. P. Arcaini, S. Bonfanti, A. Gargantini, A. Mashkoor, and E. Riccobene. Formal validation and verification of a medical software critical component. In *Formal Methods and Models for Codesign*, pages 80–89. IEEE, 2015.
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, and D. Peled. *Model checking*. MIT press, 1999.
8. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property specification patterns for finite-state verification. In *Proceedings of the second workshop on Formal methods in software practice*, pages 7–15. ACM, 1998.
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. How thorough is thorough enough? In *Correct Hardware Design and Verification Methods*, pages 65–80. Springer, 2005.
16. K. G. Larsen and B. Thomsen. A modal process logic. In *Logic in Computer Science*, pages 203–210. IEEE, 1988.
17. Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems: Specification*. Springer, 1992.
18. K. S. Namjoshi. Certifying model checkers. In *Computer Aided Verification*, pages 2–13. Springer, 2001.
19. 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.
20. 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.
21. A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science*, pages 46–57. IEEE, 1977.
22. 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.
23. L. Tan and R. Cleaveland. Evidence-based model checking. In *Computer Aided Verification*, pages 455–470. Springer, 2002.
24. J. Woodcock, P. G. Larsen, J. Bicarregui, and J. S. Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys*, 41(4), 2009.