




TOrPEDO: witnessing model correctness with topological proofs

Claudio Menghi¹ , Alessandro Maria Rizzi², Anna Bernasconi², and Paola Spoletini³

¹University of Luxembourg, Luxembourg, Luxembourg

²Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milan, Italy

³Kennesaw State University, Georgia, USA

Abstract. Model design is not a linear, one-shot process. It proceeds through refinements and revisions. To effectively support developers in generating model refinements and revisions, it is desirable to have some automated support to verify evolvable models. To address this problem, we recently proposed to adopt *topological proofs*, which are slices of the original model that witness property satisfaction. We implemented TOrPEDO, a framework that provides automated support for using topological proofs during model design. Our results showed that topological proofs are significantly smaller than the original models, and that, in most of the cases, they allow the property to be re-verified by relying only on a simple syntactic check. However, our results also show that the procedure that computes topological proofs, which requires extracting unsatisfiable cores of LTL formulae, is computationally expensive. For this reason, TOrPEDO currently handles models with a small dimension. With the intent of providing practical and efficient support for flexible model design and wider adoption of our framework, in this paper, we propose an enhanced—re-engineered—version of TOrPEDO. The new version of TOrPEDO relies on a *novel* procedure to extract topological proofs, which has so far represented the bottleneck of TOrPEDO performances. We implemented our procedure within TOrPEDO by considering Partial Kripke Structures (PKSs) and Linear-time Temporal Logic (LTL): two widely used formalisms to express models with uncertain parts and their properties. To extract topological proofs, the new version of TOrPEDO converts the LTL formulae into an SMT instance and reuses an existing SMT solver (e.g., Microsoft Z3) to compute an unsatisfiable core. Then, the unsatisfiable core returned by the SMT solver is automatically processed to generate the topological proof. We evaluated TOrPEDO by assessing (i) how does the size of the proofs generated by TOrPEDO compares to the size of the models being analyzed; and (ii) how frequently the use of the topological proof returned by TOrPEDO avoids re-executing the model checker. Our results show that TOrPEDO provides proofs that are smaller ($\approx 60\%$) than their respective initial models effectively supporting designers in creating model revisions. In a significant number of cases ($\approx 79\%$), the topological proofs returned by TOrPEDO enable assessing the property satisfaction without re-running the model checker. We evaluated our new version of TOrPEDO by assessing (i) how it compares to the previous one; and (ii) how useful it is in supporting the evaluation of alternative design choices of (small) model instances in applied domains. The results show that the new version of TOrPEDO is significantly more efficient than the previous one and can compute topological proofs for models with less than 40 states within two hours. The topological proofs and counterexamples provided by TOrPEDO are useful to support the development of alternative design choices of (small) model instances in applied domains.

Keywords: Topological proofs, Iterative design, Model checking, Theorem proving, Unsatisfiable core

1. Introduction

One of the goals of formal methods is to provide automated verification tools that support designers in producing models that satisfy a set of properties of interest. Designers benefit from automated support in two cases: (i) when their models do not satisfy the properties of interest, and (ii) when they do satisfy such properties. While model checkers provide support in the first case—by producing counterexamples that explain why properties are not satisfied—theorem provers sustain the second case—by justifying why properties are satisfied. Theorem provers usually rely on some form of deductive mechanism that, given a set of axioms, iteratively applies a set of rules until a theorem is proved (e.g., [PPZ01, PZ01]). The proof consists of the sequence of deductive rules applied to prove the theorem. Even for simple models, proving the theorem requires the use of a considerable number of deductive rules, leading to complex proofs. This makes deductive proofs difficult to understand and hardly relatable to the designer’s modeling choices. In addition, after the models are changed and model revisions are created, deductive proofs do not provide effective support for the automated verification of the model revisions.

To tackle this problem, we recently proposed the novel notion of *topological proof* (TP) [MRB20], which overcomes the complexity of deductive proofs and is designed to make proofs useful for the iterative verification of model revisions. A topological proof is a *slice* of the original model that witnesses which part of the model impacts the property satisfaction. Knowing which slice of the model impacts the property satisfaction can guide designers in refining and revising their models as it helps to select the parts of the model to be changed (or maintained). Furthermore, a topological proof can reduce the cost of formal verification. Indeed, in many cases, after a model changes, it is possible to assess the satisfaction of a property by only verifying whether the new version of the model preserves the topological proof, as opposed to re-executing the model checking procedure, which—typically—is computationally more expensive.

In our previous work [MRB20], we *formally defined topological proofs* by considering Partial Kripke Structures (PKSs) [BG99] and Linear-time Temporal Logic (LTL) to respectively express the model and the properties of interest. While it is possible to consider other modeling formalisms, we chose Partial Kripke Structures since (i) they are used in requirement elicitation to reason about system behavior from different points of view [EC01, BCE⁺06], and are a common theoretical reference language used in the formal method community for the specification of uncertain models (e.g., [GJ03, BG99, GP09, BG00]); (ii) other modeling formalisms commonly used in software development [FUMK06, Uch09], such as Modal Transition Systems [LT88] (MTSs), can be converted into Partial Kripke Structures through a simple transformation [GJ03] making our solution easily applicable to those models; and (iii) Kripke Structures (KSs) are particular instances of Partial Kripke Structures that represent complete models. As such, our definitions can also be applied to models that do not contain uncertain parts. We chose Linear-time Temporal Logic since it is a standard logic used to express properties that should hold on Partial Kripke Structures.

To support the use of topological proofs during model design, we proposed TOrPEDO (TOpological Proof drivEn Development framewOrk) [MRB20], a *novel automated verification framework*, that: (i) supports Partial Kripke Structures and Linear-time Temporal Logic; (ii) allows performing analysis and verification in the context of models in which “incompleteness” represents a conceptual uncertainty; (iii) guides refinements and revisions through complementary outputs: counterexamples and proofs; and (iv) when the system is completely specified allows understanding which changes impact or not the satisfaction of certain properties.

There exists two variants of TOrPEDO: TOrPEDO-MUP and TOrPEDO-SMT. TOrPEDO-MUP was developed in our previous work [MRB20], TOrPEDO-SMT is part of the contribution of this work.

In our previous work, we *implemented* TOrPEDO using NuSMV [CCG⁺02], i.e., an efficient and widely known model checker, and PLTL-MUP [SGT13], i.e., a tool that enables to compute a minimal subset of unsatisfiable LTL formulae from an unsatisfiable set of LTL formulae. We identify this version of TOrPEDO as TOrPEDO-MUP in the rest of the work. We evaluated TOrPEDO-MUP by considering a set of examples coming from literature including both completely specified and partially specified models. We evaluated TOrPEDO-MUP by assessing how the size of the proofs generated by TOrPEDO compares to the size of the models being analyzed (RQ1) and how frequently the use of the topological proofs returned by TOrPEDO-MUP avoids re-executing the model checker (RQ2). Our results show that topological proofs are $\approx 60\%$ smaller than the original models and that in $\approx 79\%$ of the cases topological proofs allow assessing the property satisfaction without re-executing the model checker. However, our results also show that the procedure that computes topological proofs is computationally expensive. Specifically, the procedure to extract unsatisfiable cores of LTL formulae, which is used to compute topological proofs, presents serious drawbacks. For this reason, TOrPEDO-MUP can currently handle only models of small dimensions.

Table 1. Natural language and LTL formulation of the requirements of the vacuum-cleaner robot

Textual Requirements	LTL formulae
ϕ_1 : the robot is drawing dust (<i>suck</i>) only if it has <i>reached</i> the cleaning site.	$\phi_1 \equiv \mathcal{G}(suck \rightarrow reached)$
ϕ_2 : the robot must be turned <i>on</i> before it can <i>move</i> .	$\phi_2 \equiv \mathcal{G}((\neg move) \mathcal{W} on)$
ϕ_3 : if the robot is <i>on</i> and stationary ($\neg move$), it must be drawing dust (<i>suck</i>).	$\phi_3 \equiv \mathcal{G}(((\neg move) \wedge on) \rightarrow suck)$
ϕ_4 : the robot must <i>move</i> before it is allowed to draw dust (<i>suck</i>).	$\phi_4 \equiv ((\neg suck) \mathcal{W}(move \wedge (\neg suck)))$

\mathcal{G} and \mathcal{W} are the “globally” and “weak until” LTL operators

To provide a more practical and efficient support for flexible model design, in this paper we propose a new version of TOrPEDO, hereon called TOrPEDO-SMT. TOrPEDO-SMT reduces the computational cost required to compute topological proofs. It relies on a *novel* procedure to extract topological proofs. This procedure converts LTL formulae into a Satisfiability Modulo Theories (SMT) problem instance. We reuse existing techniques to convert LTL formulae into an SMT problem [SLJ⁺06]. Specifically, since our goal is to reduce the computational cost of computing topological proofs, we implemented our translation by relying on a novel encoding [BKR15, PKRB20] based on Bit-Vectors. According to the authors, such encoding provides significantly better results when compared to other existing encodings. Then, TOrPEDO-SMT reuses an existing efficient SMT solver (e.g., Microsoft Z3 [DMB08]) to compute the unsatisfiable core of the SMT instance. Finally, the unsatisfiable core is automatically analyzed to extract the topological proof, which is returned to the model designer.

We evaluated TOrPEDO-SMT by assessing how efficient it is in analyzing models and how it compares to TOrPEDO-MUP (RQ3). TOrPEDO-SMT was able to compute topological proofs for models with less than 40 states within two hours. Furthermore, the results show that TOrPEDO-SMT is significantly more efficient than TOrPEDO-MUP. Finally, we assessed how useful TOrPEDO-SMT is in supporting the evaluation of alternative design choices of (small) model instances (RQ4). Our results show that the topological proofs and counterexamples provided by TOrPEDO effectively supported the development of a model of a small gene regulatory network.

Organization. Section 2 discusses the background. Section 3 describes TOrPEDO. Sections 4 and 5 present the theoretical results and the algorithms that support TOrPEDO. Section 6 evaluates the achieved results. Section 7 discusses related work. Section 8 presents our conclusions.

2. Running example and background notation

To illustrate TOrPEDO, we use the running example presented in Fig. 1 and Table 1; it contains a simple model describing the behavior of a vacuum-cleaner robot that has to satisfy the requirements described in Table 1. Section 2.1 introduces Partial Kripke Structures (PKS)—the modeling formalism considered in this work—and describes how the model design proceeds through refinements and revisions. Section 2.2 describes Linear-time Temporal Logic (LTL), the formalism used to express the properties of interest and its three-valued semantics. Section 2.3 shows how to model-check the satisfaction of LTL properties on PKSs.

2.1. Partial Kripke structures

The model of the vacuum-cleaner robot is represented using a *Partial Kripke Structure* (PKS) in Fig. 1a. PKSs are state machines that can be adopted when the values of given propositions on selected states is uncertain.

Definition 2.1 ([BG99],[Kri63]) A *Partial Kripke Structure* (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; $L : S \times AP \rightarrow \{\top, ?, \perp\}$ is a function that, for each state in S , associates a truth value to every atomic proposition in AP . A *Kripke Structure* (KS) M is a PKS $\langle S, R, S_0, AP, L \rangle$, where $L : S \times AP \rightarrow \{\top, \perp\}$.

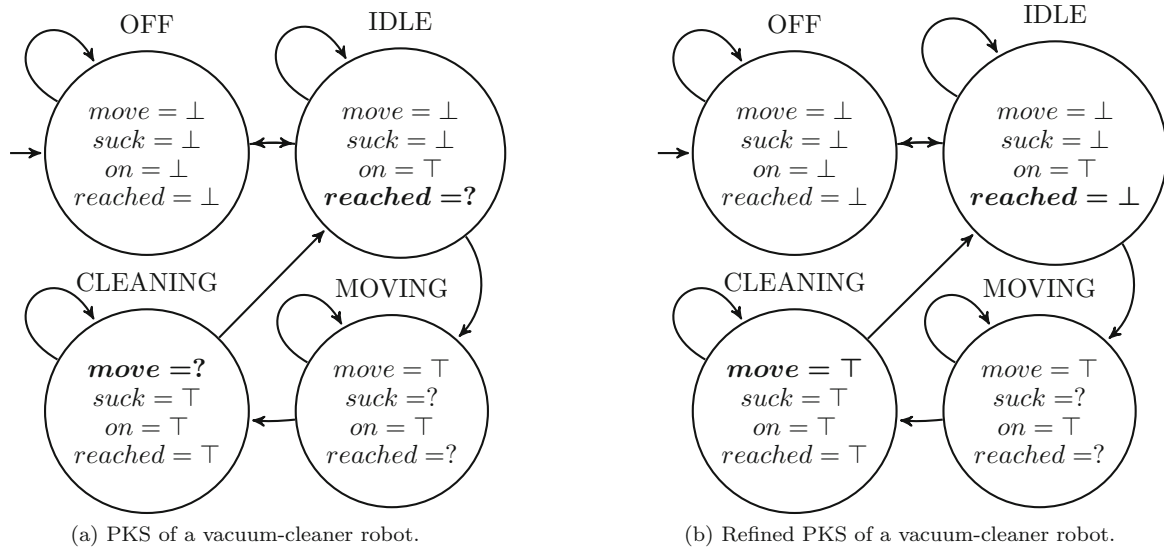


Fig. 1. Example models of a vacuum-cleaner robot

A PKS represents a system as a set of states and transitions between these states. The size $|M|$ of a PKS $M = \langle S, R, S_0, AP, L \rangle$ is $|AP| * |S| + |R| + |S_0|$. We defined the size of the PKS as the sum of (a) the number of atomic propositions assignments, that is obtained as the product between the cardinalities of the sets of the atomic propositions and the states of the PKS; (b) the cardinality of the set of the transitions of the PKS; and (c) the cardinality of the set of the initial states of the PKS. Our definition follows classical definitions used for KS (e.g., [KG96]) and ensures that the size of the PKS increases with the number of the states, the atomic propositions, and the transitions of the PKS

The PKS of the vacuum-cleaner robot presented in Fig. 1a is defined over two atomic propositions representing actions that a robot can perform: *move*, i.e., the agent travels to the cleaning site; *suck*, i.e., the agent is drawing the dust, and two atomic propositions representing conditions that can trigger actions: *on*, true when the robot is turned on; *reached*, true when the robot has reached the cleaning site. The state *OFF* represents the robot being shut down, *IDLE* the robot being turned on w.r.t. a cleaning call, *MOVING* the robot reaching the cleaning site, and *CLEANING* the robot performing its duty. Each state is labeled with the actions *move* and *suck* and the conditions *on* and *reached*. Let α and s be respectively an atomic proposition and a state. We use the notation: $\alpha = \top$ to indicate that α is true when the robot is in state s ; $\alpha = \perp$ to indicate that α is false when the robot is in state s ; $\alpha = ?$ to indicate that there is uncertainty on whether α is true or false when the robot is in state s .

Model design proceeds through *refinements* and *revisions*. We will describe how model refinements and revisions are used in incremental model design in Sect. 3. Refinements assign either a \top or a \perp value to an atomic proposition α , in a state s , such that $L(s, \alpha) = ?$.

Definition 2.2 ([BG00]) Let $M = \langle S, R, S_0, AP, L \rangle$ be a PKS. A *refinement* of M is a PKS $M_{rf} = \langle S, R, S_0, AP, L_{rf} \rangle$ where L_{rf} is such that

- for all $s \in S$, $\alpha \in AP$ if $L(s, \alpha) = \top \rightarrow L_{rf}(s, \alpha) = \top$;
- for all $s \in S$, $\alpha \in AP$ if $L(s, \alpha) = \perp \rightarrow L_{rf}(s, \alpha) = \perp$.

We use the notation $M \preceq M_{rf}$ to indicate that M_{rf} is a refinement of M . For example, the PKS in Fig. 1b is a refinement of the PKS in Fig. 1a, obtained by changing the value of the proposition *reached* into \perp in state *IDLE* and the value of the proposition *move* into \top in state *CLEANING*.

Definition 2.3 ([BG00]) Let M be a PKS and let M_{comp} be a KS. Then, M_{comp} is a *completion* [BG00] of M if and only if M_{comp} is a refinement of M .

Intuitively, a completion of a PKS is a refinement of the PKS obtained by replacing all the uncertain values (?) assigned to the atomic propositions by \perp or \top .

During a revision, a designer can add and remove states and transitions and/or change the values of the atomic propositions in the states of the PKS.

Definition 2.4 Let $M = \langle S, R, S_0, AP, L \rangle$ and $M_{rv} = \langle S_{rv}, R_{rv}, S_{rv,0}, AP_{rv}, L_{rv} \rangle$ be two PKSs. Then, M_{rv} is a revision of M if and only if $AP \subseteq AP_{rv}$.

This definition of revision requires the model M_{rv} to contain at least the same propositions of M , i.e., $AP \subseteq AP_{rv}$. However, the definition does not force any relation between S, R, S_0, L and $S_{rv}, R_{rv}, S_{rv,0}, L_{rv}$, respectively. Therefore, a revision M_{rv} can be obtained from M by arbitrarily changing its initial states, states, and transitions. Intuitively, this means that the only constraint the designer has to respect during a revision is not to remove propositions from the set of atomic propositions. This condition is necessary to ensure that any property that can be evaluated on M can also be evaluated on M_{rv} , i.e., every atomic proposition has a value in each of the states of the PKS. Instead, the deactivation of a proposition can be simulated by associating its value to \perp in all the states of M_{rv} .

Lemma 2.1 Let $M = \langle S, R, S_0, AP, L \rangle$ be a PKS and let $M_{rf} = \langle S, R, S_0, AP, L_{rf} \rangle$ be a refinement of M . Then, M_{rf} is a revision of M .

A refinement is a particular type of revision; for example, the PKS in Fig. 1b is an example of a revision of the PKS in Fig. 1a.

Proof Sketch Since the PKSs M and M_{rf} share the atomic proposition set AP , the condition $AP \subseteq AP_{rv}$ of Definition 2.4 is satisfied. \square

2.2. Linear-time temporal logic and three-valued semantics

The properties that the model of the vacuum-cleaner robot should satisfy are expressed in *Linear-time Temporal Logic* (LTL) in Table 1.

LTL formulae combine atomic propositions with the Boolean connectors “and” (\wedge) and “not” (\neg) and the temporal modalities “next” (\mathcal{X}) and “until” (\mathcal{U}).

Definition 2.5 ([Pnu77]) Given a set AP of atomic propositions (with $p \in AP$), an LTL formula ϕ is formed according to the following grammar:

$$\phi = true \mid p \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \mathcal{X}\phi \mid \phi_1 \mathcal{U} \phi_2 \quad (1)$$

where ϕ_1 and ϕ_2 are LTL formulae.

The “or” (\vee), “implication” (\rightarrow) and “equivalence” (\leftrightarrow) Boolean operators are derived using the operators “and” (\wedge) and “not” (\neg). Furthermore, the temporal operators “eventually” (\mathcal{F}), “globally” (\mathcal{G}) and “weak until” (\mathcal{W}), are derived using the other temporal operators as usual. For example, $\phi_1 \mathcal{W} \phi_2$ is defined using the “until” (\mathcal{U}) and the “globally” (\mathcal{G}) operators as $(\phi_1 \mathcal{U} \phi_2) \vee \mathcal{G} \phi_1$.

For KSs, we consider the classical LTL semantics $[M \models \phi]$ over infinite words, which associates to a model M and a formula ϕ a truth value in the set $\{\perp, \top\}$ (see for example [BK08]). For PKS, instead, the *three-valued LTL semantics* [BG99] $[M \models \phi]$ associates to a model M and a formula ϕ a truth value in the set $\{\perp, ?, \top\}$, being based on the information ordering $\top > ? > \perp$. The three-valued LTL semantics is defined by considering paths on the model M . A path π is an infinite sequence of states s_0, s_1, \dots such that, for all $i \geq 0$, $(s_i, s_{i+1}) \in R$. We use the notation π^i to indicate the infinite sub-sequence of π that starts at position i , and $Path(s)$ to indicate the set of paths that start in the state s .

Definition 2.6 ([BG99]) Let $M = \langle S, R, S_0, AP, L \rangle$ be a PKS, $\pi = s_0, s_1, \dots$ be a path, and ϕ be an LTL formula. Then, the *three-valued semantics* $[(M, \pi) \models \phi]$ is defined inductively as follows:

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

Let $M = \langle S, R, S_0, AP, L \rangle$ be a PKS, and ϕ be an LTL formula. Then $[M \models \phi] = \min(\{(M, \pi) \models \phi \mid \pi \in \text{Path}(s) \text{ and } s \in S_0\})$.

The *comp* operator maps \top to \perp , \perp to \top , and $?$ to $?$. The minimum and the maximum functions are defined by considering the order $\top > ? > \perp$. The minimum and the maximum functions are extended to sets by considering $\min(\emptyset) = \top$ and $\max(\emptyset) = \perp$.

For example, consider the path *OFF, IDLE, MOVING^ω* of the PKS in Fig. 1a, where *MOVING^ω* indicates that the state *MOVING* is entered infinitely often. The three-valued semantics associates to the LTL property ϕ_1 the value $?$, since its satisfaction depends on the value assigned to the proposition *reached* in the states *IDLE* and *MOVING*.

2.3. Model checking

Checking KSs with respect to LTL properties can be done by using classical model checking procedures. We assume that the function CHECK returns a tuple $\langle res, c \rangle$, where *res* is the model checking result in $\{\top, \perp\}$ and *c* is the counterexample if *res* = \perp , else an empty sequence. Note that the model checking problem of a property ϕ on a KS *M* can be reduced to the satisfiability problem of the LTL formula $\Phi_M \wedge \neg\phi$, where Φ_M represents the behaviors of model *M*. If $\Phi_M \wedge \neg\phi$ is satisfiable, then $[M \models \phi] = \perp$, otherwise $[M \models \phi] = \top$.

Checking a PKS *M* with respect to an LTL property ϕ considering the three-valued semantics is done by performing twice the classical model checking procedure for KSs [BG00], one considering an optimistic approximation M_{opt} and one considering a pessimistic approximation M_{pes} . These two procedures consider the LTL formula $\phi_{neg} = F(\phi)$, where *F* transforms ϕ with the following steps: (i) negation of ϕ ; (ii) conversion of $\neg\phi$ in negation normal form;¹ (iii) replacement of every subformula $\neg\alpha$, where α is an atomic proposition, with a new proposition $\bar{\alpha}$.

To create the optimistic and pessimistic approximations M_{opt} and M_{pes} , the PKS $M = \langle S, R, S_0, AP, L \rangle$ is first converted into its *complement-closed* version $M_c = \langle S, R, S_0, AP_c, L_c \rangle$ where the set of atomic propositions $AP_c = AP \cup \overline{AP}$ is such that $\overline{AP} = \{\bar{\alpha} \mid \alpha \in AP\}$. Atomic propositions in \overline{AP} are called *complement-closed* propositions. Function L_c is such that, for all $s \in S$ and $\alpha \in AP$, $L_c(s, \alpha) = L(s, \alpha)$ and, for all $s \in S$ and $\bar{\alpha} \in \overline{AP}$, $L_c(s, \bar{\alpha}) = \text{comp}(L(s, \alpha))$. The complement-closed PKS of the vacuum-cleaner agent in Fig. 1a presents eight propositional assignments in the state *IDLE*: *move* = \perp , $\overline{\text{move}}$ = \top , *suck* = \perp , $\overline{\text{suck}}$ = \top , *on* = \top , $\overline{\text{on}}$ = \perp , *reached* = $?$, and $\overline{\text{reached}}$ = $?$.

The two model checking runs for a PKS $M = \langle S, R, S_0, AP, L \rangle$ are based respectively on an optimistic ($M_{opt} = \langle S, R, S_0, AP_c, L_{opt} \rangle$) and a pessimistic ($M_{pes} = \langle S, R, S_0, AP_c, L_{pes} \rangle$) approximation of *M*'s related complement-closed version $M_c = \langle S, R, S_0, AP_c, L_c \rangle$. Function L_{pes} (resp. L_{opt}) is such that

- for all $s \in S, \alpha \in AP_c$, and $L_c(s, \alpha) \in \{\top, \perp\}$, it holds that $L_{pes}(s, \alpha) = L_c(s, \alpha)$ (resp. $L_{opt}(s, \alpha) = L_c(s, \alpha)$), and
- for all $s \in S, \alpha \in AP_c$, and $L_c(s, \alpha) = ?$, it holds that $L_{pes}(s, \alpha) = \perp$ (resp. $L_{opt}(s, \alpha) = \top$).

Let \mathcal{A} be a KS and ϕ be an LTL formula, $\mathcal{A} \models^* \phi$ is true if \mathcal{A} does not contain any path that satisfies the formula $F(\phi)$.

Theorem 2.1 ([BG99]) Let ϕ be an LTL formula, let $M = \langle S, R, S_0, AP, L \rangle$ be a PKS, and let M_{pes} and M_{opt} be the pessimistic and optimistic approximations of *M*'s relative complement-closed M_c . Then

$$[M \models \phi] \stackrel{\text{def}}{=} \begin{cases} \top & \text{if } M_{pes} \models^* \phi \\ \perp & \text{if } M_{opt} \not\models^* \phi \\ ? & \text{otherwise} \end{cases} \quad (2)$$

¹ An LTL formula ϕ is in *negation normal form* if negations are applied only to atomic propositions. Conversion of an LTL formula into its negation normal form can be achieved by pushing negations inward and replacing them with their duals—for details see [BK08].

We call CHECK^* the function that computes the result of the operator \models^* . Three-valued model checking tools take as input the KS M and the LTL property ϕ , and return a tuple $\langle res, c \rangle$, where res is the model checking result in $\{\top, ?, \perp\}$, and c can be:

- an empty sequence (when M satisfies ϕ),
- a *definitive-counterexample* (when M violates ϕ), or
- a *possible-counterexample* (when M possibly-satisfies ϕ).

Both counterexamples indicate behaviors that violate the properties of interest: *definitive-counterexamples* depend on already performed design choices, *possible-counterexamples* are based also on uncertain actions and conditions. Intuitively, in the construction of the optimistic completion M_{opt} , the algorithm “tries 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. The presence of a violating behavior in M_{opt} can be checked by verifying whether $\Phi_{M_{opt}} \wedge \phi_{neg}$ is satisfiable, where $\Phi_{M_{opt}}$ represents the behaviors of model M_{opt} and $\phi_{neg} = \text{F}(\phi)$. If $\Phi_{M_{opt}} \wedge \phi_{neg}$ is satisfiable, then there exists a behavior that satisfies ϕ_{neg} , that is it violates ϕ . Therefore, such behavior is a definitive counterexample. Viceversa, in the construction of the pessimistic completion M_{pes} , the three-valued model checker “tries its best” to construct a KS that violates ϕ . If no violating behavior is found in M_{pes} , then $M \models \phi$. The presence of a violating behavior in M_{pes} can be checked by verifying whether $\Phi_{M_{pes}} \wedge \phi_{neg}$ is satisfiable, where $\Phi_{M_{pes}}$ represents the behaviors of model M_{pes} . If $\Phi_{M_{pes}} \wedge \phi_{neg}$ is not satisfiable, then it does not exist any behavior that satisfies ϕ_{neg} , that is ϕ is satisfied. Otherwise, it could be the case where there exists some completion in which ϕ holds and others in which it does not hold. In this case, the three-valued model checker returns $?$.

For example, the PKS represented in Fig. 1a possibly satisfies LTL property ϕ_1 . The path $OFF, IDLE, MOVING^\omega$ is a possible-counterexample for this property.

While definitive and possible counterexamples provide information when the property of interest is violated, or possibly violated, they do not provide any information that explains why the model satisfies or possibly satisfies a property of interest. To tackle this problem, we propose TOrPEDO.

3. The topological proof driven development framework

The TOpological Proof drivEN Development framewOrk (TOrPEDO) is a development framework which supports the use of *topological proofs* (TPs) during the model design. The TOrPEDO framework is illustrated in Fig. 2 and carries out verification in four phases: INITIAL DESIGN, ANALYSIS, REVISION, and RE-CHECK.

Initial design (1). The model M of the system is expressed using a PKS (1), which can be generated from other languages (e.g., MTS), along with the property of interest ϕ , expressed using LTL (2).

Running example. The model in Fig. 1a contains an example of PKS with an initial design for the vacuum-cleaner robot. Table 1 contains the properties of interests for this model.

Analysis (2). TOrPEDO provides automated analysis support, which includes the following elements:

- (i) Information about *what is wrong* in the current model design. This information includes a definitive-counterexample (3 \perp -CE), which can be used to produce a revised version M_{rv} of M that satisfies or possibly satisfies the property of interest.
- (ii) Information about *what is correct* in the current design. This information includes definitive-topological proofs (4 \top -TP) that indicate a portion of the design that ensures satisfaction of the property.
- (iii) Information about *what could be wrong/correct* in the current design, depending on how uncertainty is removed. This information includes: a possible-counterexample (5 $?$ -CE) and a possible-topological proof (6 $?$ -TP), indicating a portion of the design that ensures the possible satisfaction of the property of interest.

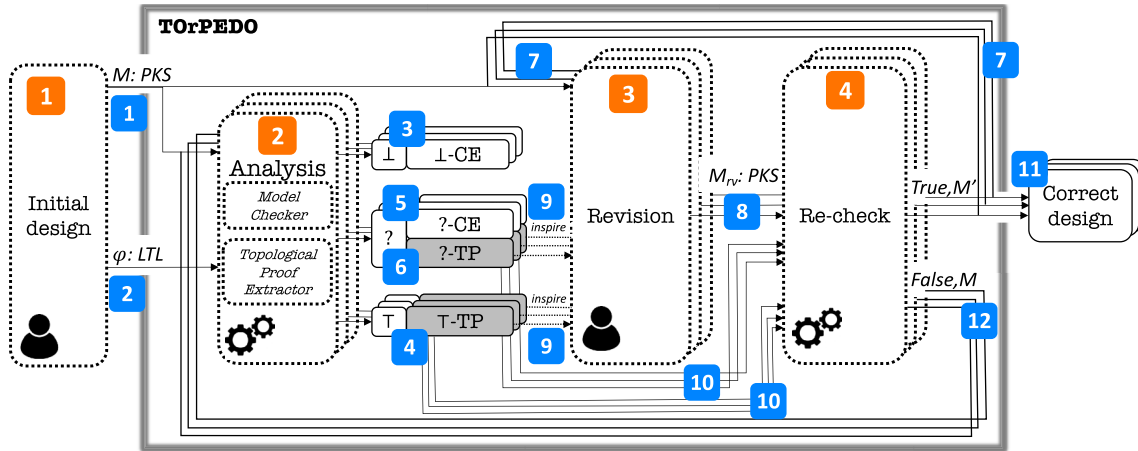


Fig. 2. Phases of the TORPEDO development framework. Continuous arrows represent inputs and outputs to phases. Numbers are used to reference the image in the text

Table 2. Results provided by TORPEDO for properties ϕ_1 , ϕ_2 , ϕ_3 and ϕ_4 . \top , \perp and $?$ indicate that the property is satisfied, violated and possibly satisfied.

		$?-CE$	$OFF, IDLE, (MOVING)^\omega$.
ϕ_1	$?$	$?-TP$	$\begin{cases} \text{TTP: } \langle CLEANING, reached, \top \rangle, \langle OFF, suck, \perp \rangle, \langle IDLE, suck, \perp \rangle, \langle MOVING, suck, ? \rangle \\ \text{TPT: } \langle OFF, \{OFF, IDLE\} \rangle, \langle IDLE, \{OFF, IDLE, MOVING\} \rangle, \\ \langle MOVING, \{MOVING, CLEANING\} \rangle, \langle CLEANING, \{CLEANING, IDLE\} \rangle \\ \text{TPI: } \{\{OFF\}\} \end{cases}$
ϕ_2	\top	$\top-TP$	$\begin{cases} \text{TTP: } \langle MOVING, on, \top \rangle, \langle CLEANING, on, \top \rangle, \langle OFF, move, \perp \rangle, \langle IDLE, move, \perp \rangle \\ \text{TPT: } \langle OFF, \{OFF, IDLE\} \rangle, \langle IDLE, \{OFF, IDLE, MOVING\} \rangle, \\ \langle MOVING, \{MOVING, CLEANING\} \rangle, \langle CLEANING, \{CLEANING, IDLE\} \rangle \\ \text{TPI: } \{\{OFF\}\} \end{cases}$
ϕ_3	\perp	$\perp-CE$	$OFF, IDLE^\omega$
		$?-CE$	$OFF, (IDLE, MOVING, CLEANING, IDLE, OFF)^\omega$
ϕ_4	$?$	$?-TP$	$\begin{cases} \text{TTP: } \langle OFF, suck, \perp \rangle, \langle IDLE, suck, \perp \rangle, \langle MOVING, suck, ? \rangle, \langle MOVING, move, \top \rangle \\ \text{TPT: } \langle OFF, \{OFF, IDLE\} \rangle, \langle IDLE, \{OFF, IDLE, MOVING\} \rangle \\ \text{TPI: } \{\{OFF\}\} \end{cases}$

The automated analysis support relies on two components: the *model checker* and the *topological proof extractor*. The model checker verifies whether a property is satisfied or violated by the current model and is implemented by reusing existing algorithms from the literature, such as the one presented in Sect. 2. The topological proof extractor computes the topological proof and is discussed in Sect. 3.1.

In the following we will use the notation x -topological proofs or x -TP to respectively indicate definitive-topological or possible-topological proofs.

Running example. The results returned by TORPEDO for the different properties in our motivating example are presented in Table 2. Property ϕ_2 is satisfied, ϕ_3 is not. In those cases, TORPEDO returns respectively a definitive-proof and a definitive-counterexample. Since ϕ_1 and ϕ_4 are possibly satisfied, in both cases a possible-counterexample and a possible-topological proof are returned. For ϕ_1 , the possible-counterexample shows a run that may violate the property of interest. The possible-topological proof for ϕ_1 in Table 2 shows that if OFF remains the only initial state (TPI), *reached* still holds in *CLEANING*, and *suck* does not hold in *OFF* and *IDLE*, while unknown in *MOVING* (TTP), property ϕ_1 remains possibly satisfied. In addition, all transitions must be preserved (TPT).² Note that the proof highlights portions of the model that influence the property satisfaction. For example, by inspecting the proof, the designer understands that she can change the value of the proposition *reached* in all the states of the PKS, with the exception of the state *CLEANING*, without making the property violated.

² The precise formal descriptions of x -topological proofs, TPI, TPT and TTP are presented in Sect. 4.

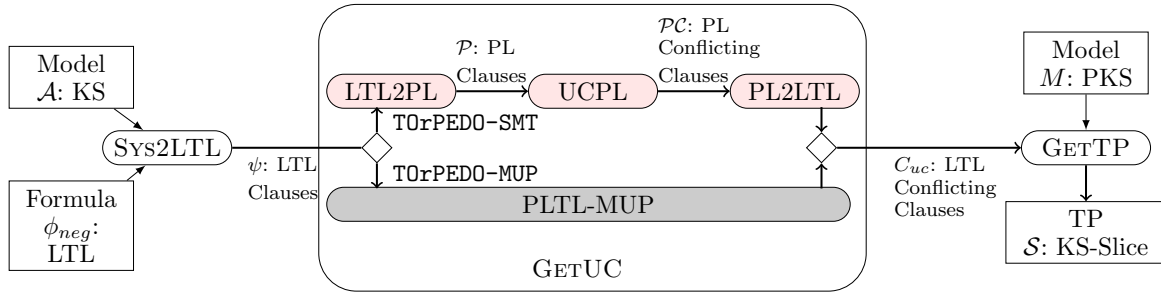


Fig. 3. Components of the Topological Proof Extractor of TOrPEDO-MUP and TOrPEDO-SMT.

Revision (3). Revisions (8)—see Definition 2.4) can be obtained by changing some parts of the model: adding/removing states and transitions or by changing propositions labelling inside states, and are defined by considering the TP (9).

Running example. The designer may want to propose a revision that still does not violate properties ϕ_1 , ϕ_2 , and ϕ_4 . Thus, she changes the values of some atomic propositions: *move* becomes \top in state *CLEANING* and *reached* becomes \perp in state *IDLE*. Fig. 1b contains the revision of the PKS in Fig. 1a obtained by applying these changes. Since ϕ_1 , ϕ_2 , and ϕ_4 were previously not violated, TOrPEDO performs the RE-CHECK phase for each property.

Re-check (4). The automated verification tool provided by TOrPEDO checks whether all the changes in the current model revision are compliant with the x -TPs (10), i.e., changes applied to the revised model do not include parts that had to be preserved according to the x -topological proof. If a property of interest is (possibly) satisfied in a previous model, and the revision of the model is compliant with the property x -TP, the designer has the guarantee that the property is (possibly) satisfied in the revision. Thus, she can perform another model revision round (7) or approve the current design (11). Otherwise, TOrPEDO re-executes the ANALYSIS (12).

Running example. In the vacuum-cleaner case, the revision in Fig. 1b passes the RE-CHECK and the designer proceeds to a new revision phase.

3.1. Topological proof extractor

We present two implementations for the topological proof extractor component: our previous implementation [MRB20], which is based on PLTL-MUP [SGT13], and a novel SMT-based procedure, which is part of the contribution of this work. The versions of TOrPEDO that use these two topological proof extractor components are named TOrPEDO-MUP and TOrPEDO-SMT. In the rest of this work, we will use TOrPEDO-MUP and TOrPEDO-SMT when referring to the specific solver used to extract topological proofs, TOrPEDO when indicating the general method.

The topological proof extractor employed during the ANALYSIS phase takes as input a PKS M , its optimistic or pessimistic approximation \mathcal{A} and an LTL formula $\phi_{neg} = F(\phi)$ (see Sect. 2.3) and returns a slice S of the KS. Figure 3 conceptually describes the components of the topological proof extractor used in both TOrPEDO-MUP and TOrPEDO-SMT: rectangular boxes with sharp corners represent the inputs and outputs of the topological proof extractor; rectangular boxes with rounded corners represent software components. Their background color indicates whether the components are part of TOrPEDO-MUP (i.e., gray), TOrPEDO-SMT (i.e., red), or both (i.e., white). The labels on the arrows describe the type of inputs and outputs of each component.

The components implement the steps of Algorithm 1: TOrPEDO employs the Sys2LTL procedure to convert the KS \mathcal{A} and the LTL formula ϕ_{neg} into a set C of LTL clauses (Line 2). The set of LTL clauses C contains: i) clauses $C_{\mathcal{A}}$ encoding the behaviors of the model \mathcal{A} and ii) the formula ϕ_{neg} . Since ϕ is satisfied, none of the behaviors of the model satisfy ϕ_{neg} ; since the behaviors that violate the property cannot occur, some of the clauses in $C_{\mathcal{A}}$ must conflict with ϕ_{neg} . Our GETUC procedure detects such clauses and stores them within the set of conflicting clauses C_{uc} (Line 3). Finally, the GETTP component maps the conflicting clauses within C_{uc} into the corresponding slice S of the KS \mathcal{A} (Line 4).

Algorithm 1 Compute Topological Proofs.

```

1: function CTP_KS( $M, \mathcal{A}, \phi_{neg}$ )
2:    $C = \text{SYS2LTL}(\mathcal{A}, \phi_{neg})$ 
3:    $C_{uc} = \text{GETUC}(C)$ 
4:    $\mathcal{S} = \text{GETTP}(M, \mathcal{A}, C_{uc})$ 
5:   return  $\mathcal{S}$ 
6: end function

```

TORPEDO-MUP and TORPEDO-SMT differ in the implementation of the GETUC procedure. TORPEDO-MUP implements it by using PLTL-MUP [SGT13], which returns the conflicting clauses (C_{uc}). TORPEDO-SMT, instead, translates the LTL clauses into a set of clauses \mathcal{P} in propositional logic (PL), maintaining a map between each LTL clause and the propositional logic clauses generated from it. Then, it exploits existing SMT-based solvers to compute the conflicting propositional logic clauses (\mathcal{PC}). Finally, it uses the previously built map to detect the LTL clauses that generated the conflicting propositional logic clauses.

The two procedures are described in detail in Sects. 5.3 and 5.4. We evaluate TORPEDO-MUP and TORPEDO-SMT in Sect. 6.

4. Topological proofs: a formal definition

In this section, we introduce the notion of *topological proof*. The pursued proof consists of a set of clauses that specify certain topological properties of M ; these represent the portion of the model that explains how it satisfies (or possibly-satisfies) the imposed claim. Different kinds of clauses are defined next.

Definition 4.1 Let $M = \langle S, R, S_0, AP, L \rangle$ be a PKS. A *Topological Proof clause* (TP-clause) γ for M is either:

- a *Topological Proof Propositional clause* (TPP-clause), i.e., a triad $\langle s, \alpha, v \rangle$ where $s \in S$, $\alpha \in AP$, and $v \in \{\top, ?, \perp\}$;
- a *Topological Proof Transitions-from-state clause* (TPT-clause), i.e., a pair $\langle s, T \rangle$, such that $s \in S$, $T \subseteq S$;
- a *Topological Proof Initial-states clause* (TPI-clause), i.e., an element $\langle S_0 \rangle$.

These clauses indicate *topological properties* of a PKS M . Informally, TPP-clauses constrain how states are labeled (L), TPT-clauses constrain how states are connected (R), and TPI-clauses constrain the initial states of the model (S_0). For example, let us consider in Table 2 the proof obtained for property ϕ_1 :

- $\langle \text{CLEANING}, \text{reached}, \top \rangle$ is a TPP-clause that constrains the atomic proposition *reached* to be labeled as true (\top) in the state *CLEANING*;
- $\langle \text{OFF}, \{\text{OFF}, \text{IDLE}\} \rangle$ is a TPT-clause that constrains the transition from *OFF* to *OFF* and from *OFF* to *IDLE* to not be removed; and
- $\langle \{\text{OFF}\} \rangle$ is a TPI-clause that constrains the state *OFF* to remain the initial state of the system.

We say that a state s_i is constrained by a TPP-clause $\langle s, \alpha, v \rangle$ if $s = s_i$, by a TPT-clause $\langle s, T \rangle$ if $s = s_i$ or $s_i \in T$, and by a TPI-clause $\langle S_0 \rangle$ if $s_i \in S_0$.

We now define the notion of Ω -related PKS that is then used to formally define a topological proof.

Definition 4.2 Let $M = \langle S, R, S_0, AP, L \rangle$ be a PKS and Ω be a set of TP-clauses for M . Then, a PKS Ω -related to M is a PKS $M_{\Omega rel} = \langle S_{\Omega rel}, R_{\Omega rel}, S_{\Omega rel,0}, AP_{\Omega rel}, L_{\Omega rel} \rangle$, such that the following conditions hold:

- i) $AP \subseteq AP_{\Omega rel}$;
- ii) for every TPP-clause $\langle s, \alpha, v \rangle \in \Omega$, the condition $s \in S_{\Omega rel}$ and $v = L_{\Omega rel}(s, \alpha)$ holds;
- iii) for every TPT-clause $\langle s, T \rangle \in \Omega$, the condition $s \in S_{\Omega rel}$, $T \subseteq S_{\Omega rel}$, and $T = \{s_i \in S_{\Omega rel} \mid (s, s_i) \in R_{\Omega rel}\}$ holds;

iv) for every TPI-clause $\langle S_0 \rangle \in \Omega$, the condition $S_0 = S_{\Omega_{rel},0}$ holds.

Intuitively, a PKS Ω -related to M is a PKS $M_{\Omega_{rel}}$ that is compliant with the set of TP-clauses Ω for M . Specifically:

- i) the atomic propositions in the set AP of M are also included in the set $AP_{\Omega_{rel}}$ of $M_{\Omega_{rel}}$;
- ii) the values of the atomic propositions in the TPP-clauses are not changed;
- iii) the states of M that are constrained by a TPT-clause are also part of $M_{\Omega_{rel}}$. The set of the outgoing transitions T of a state s of M that are part of a TPT-clause $\langle s, T \rangle$ are also the outgoing transitions of s in $M_{\Omega_{rel}}$. Note that the states of M that are not constrained by a TPT-clause are not necessarily states of $M_{\Omega_{rel}}$; and
- iv) the initial states of M that are in a TPI-clause are exactly the initial states of $M_{\Omega_{rel}}$.

Based on these observations, a topological proof is then defined as follows.

Definition 4.3 Let $M = \langle S, R, S_0, AP, L \rangle$ be a PKS, let ϕ be an LTL property, let Ω be a set of TP-clauses for M , and let x be a truth value in $\{\top, ?\}$. A set of TP-clauses Ω is an x -topological proof (or x -TP) for ϕ in M if: (i) $[M \models \phi] = x$; and (ii) every PKS $M_{\Omega_{rel}}$ Ω -related to M is such that $[M_{\Omega_{rel}} \models \phi] \geq x$.

The operator \geq assumes that values $\top, ?, \perp$ are ordered considering the classical information ordering $\top > ? > \perp$ among the truth values [BG99]. Intuitively, an x -topological proof for M ensures that every Ω -related PKS $M_{\Omega_{rel}}$ to M satisfies the property ϕ “at least as much as M does”. We call \top -TP a *definitive-topological proof* and $?$ -TP a *possible-topological proof*. Intuitively, a definitive-topological proof for M ensures that every Ω -related PKS $M_{\Omega_{rel}}$ to M satisfies the property ϕ . A possible-topological proof for M ensures that every Ω -related PKS $M_{\Omega_{rel}}$ to M satisfies or possibly satisfies the property ϕ .

The size of an x -topological proof Ω is defined as $|\Omega| = \sum_{c \in \Omega} |c|$ where:

- $|c| = 1$ if $c = \langle s, \alpha, v \rangle$;
- $|c| = |T|$ if $c = \langle s, T \rangle$; and
- $|c| = |S_0|$ if $c = \langle S_0 \rangle$.

Note that, for the PKS in Fig. 1a, Table 2 shows two $?$ -TPs for properties ϕ_1 and ϕ_4 (respectively of size 14 and 10), and one \top -TP for property ϕ_2 (of size 14).

To introduce the core property of topological proofs, we first present the notion of Ω_x -revision, a revision of the model that follows the constraints imposed by the topological proof. Note that, differently from the notion of Ω -relation, which is defined by considering an arbitrary set of TP-clauses, the notion of Ω_x -revision is defined by considering the topological proof, which is a set of TP-clauses that follows the conditions specified in Definition 4.3.

Definition 4.4 Let M and $M_{\Omega_{rv}}$ be two PKSs, let ϕ be an LTL property, and let Ω be an x -TP for ϕ in M . Then, $M_{\Omega_{rv}}$ is an Ω_x -revision of M if $M_{\Omega_{rv}}$ is Ω -related to M .

The Ω_x -revision $M_{\Omega_{rv}}$ of M is such that $M_{\Omega_{rv}}$ is Ω -related to M ; this means that it is obtained by changing the model M while preserving the statements that are specified in the x -TP for ϕ . A revision $M_{\Omega_{rv}}$ of M is *compliant* with the x -TP for a property ϕ in M if it is an Ω_x -revision of M . Intuitively, a Ω_x -revision of M is a PKS $M_{\Omega_{rv}}$ obtained from M by changing any topological aspect that does not impact on the set of TP-clauses Ω . Specifically, models can be changed as follows:

1. any transition whose source state is not the source state of a transition included in the TPT-clauses can be added or removed from the PKS;
2. any value of a proposition that is not constrained by a TPP-clause can be changed;
3. states can be added and removed if they are not constrained by any TPT-, TPP-, or TPI-clause;
4. initial states cannot be changed if Ω contains a TPI-clause.

For example, consider the topological proof for the property ϕ_1 presented in Table 2. The PKS in Fig. 1a is an $\Omega_?$ -revision of the PKS in Fig. 1b since the values of the atomic propositions *reached* in *IDLE* and *move* in *CLEANING* are not constrained by any TPP-clause.

In the following, we claim and prove that, if M is a PKS that satisfies the property ϕ , then any Ω_{\top} -revision of M also satisfies ϕ . In other words, if the model satisfies ϕ and it is revised in accordance with its topological

proof, then Ω_{\top} -revisions of the model are generated and the revised model also satisfies ϕ . If M is a PKS that possibly satisfies the property ϕ , then any $\Omega_{?}$ -revision possibly satisfies or satisfies ϕ . This means that, if the model possibly satisfies ϕ and is revised considering its topological proof, then $\Omega_{?}$ -revisions of the model are generated and these also satisfy/possibly-satisfy ϕ . Consequently, when the model is revised according to its topological proof, there is no need to run the model checker to verify that ϕ is not satisfied on the revised model.

Theorem 4.1 Let M be a PKS, let ϕ be an LTL property such that $[M \models \phi] = \top$, and let Ω be a \top -TP for ϕ in M . Then every Ω_{\top} -revision $M_{\Omega_{rv}}$ is such that $[M_{\Omega_{rv}} \models \phi] = \top$.
Let M be a PKS, let ϕ be an LTL property such that $[M \models \phi] = ?$, and let Ω be a $?$ -TP for ϕ in M . Then every $\Omega_{?}$ -revision $M_{\Omega_{rv}}$ is such that $[M_{\Omega_{rv}} \models \phi] \in \{\top, ?\}$.

Proof Sketch We prove the first statement of the Theorem; the proof of the second statement is obtained by following the same steps.

The theorem assumes that $M_{\Omega_{rv}}$ is an Ω_{\top} -revision of M . If Ω is a \top -TP for ϕ in M , then $M_{\Omega_{rv}}$ is Ω -related to M (by Definition 4.4). Since Ω is a \top -TP for ϕ in $M_{\Omega_{rv}}$ and $M_{\Omega_{rv}}$ is Ω -related to M , then $[M_{\Omega_{rv}} \models \phi] \geq \top$ (by Definition 4.3). \square

5. Automated support

This section describes the algorithms that support TOrPEDO. Section 5.1 describes how the model checker and the topological proof extractor are integrated into the ANALYSIS phase (2) of TOrPEDO. Section 5.2 describes in detail the steps of TOrPEDO used to compute topological proofs (Algorithm 1). Section 5.3 describes the implementation used by TOrPEDO-MUP for computing topological proofs. Section 5.4 describes the implementation of the SMT-based algorithm used by TOrPEDO-SMT for computing topological proofs. Section 5.5 presents the implementation of the RE-CHECK (4) phase of TOrPEDO.

5.1. Analysis

To analyze a PKS $M = \langle S, R, S_0, AP, L \rangle$ (1), TOrPEDO uses the three-valued model checking framework based on Theorem 2.1. The model checking result is provided as output by the ANALYSIS phase of TOrPEDO, whose behavior is described in Algorithm 2.

The algorithm returns a tuple $\langle x, y \rangle$, where x is the verification result and y is a set containing the counterexample, the topological proof or both of them. The algorithm first checks whether the optimistic approximation M_{opt} of the PKS M satisfies property ϕ (2, Line 2). For computing M_{opt} our implementation follows the steps described in Sect. 2.3: it first computes the complement-closed PKS by computing the set of complement-closed atomic propositions and the function L_c , and then, it computes the optimistic approximation, by calculating the functions L_{opt} . We used the NuSMV model checker for implementing the CHECK* procedure. If the optimistic approximation M_{opt} of the PKS M violates the property ϕ , the property is violated by the PKS and the definitive-counterexample c_d (3, \perp -CE) is returned (Line 3). Otherwise, the algorithm checks whether the pessimistic approximation M_{pes} of the PKS M satisfies property ϕ (Line 5). As for the case of M_{opt} , for computing M_{pes} our implementation follows the steps described in Sect. 2.3. If the PKS M satisfies property ϕ , the property is satisfied and the value \top is returned along with the definitive-topological proof (4, \top -TP) computed by the CTP_KS procedure applied on the pessimistic approximation M_{pes} and the property ϕ_{neg} (Line 7). Otherwise, the property is possibly satisfied and the value $?$ is returned along with the possible-counterexample c_p (5, $?$ -CE) and the possible-topological proof (6, $?$ -TP) computed by the CTP_KS procedure applied to M_{opt} and ϕ_{neg} (Line 9).

Algorithm 2 The ANALYSIS algorithm.

```

1: function ANALYZE( $M, \phi$ )
2:    $\langle res_{opt}, c_d \rangle = \text{CHECK}^*(M_{opt}, \phi)$ 
3:   if  $res == \perp$  then return  $\langle \perp, \{c_d\} \rangle$ 
4:   else
5:      $\langle res_{pes}, c_p \rangle = \text{CHECK}^*(M_{pes}, \phi)$ 
6:     if  $res_{pes} == \top$  then
7:       return  $\langle \top, \{\text{CTP\_KS}(M, M_{pes}, \phi_{neg})\} \rangle$ 
8:     else
9:       return  $\langle ?, \{c_p, \text{CTP\_KS}(M, M_{opt}, \phi_{neg})\} \rangle$ 
10:    end if
11:  end if
12: end function

```

Table 3. Rules to transform the KS in LTL formulae.

$$c_i = \bigvee_{s \in S_0} p(s)$$

The KS is initially in one of its initial states.

$$C_N = \{\mathcal{G}(\neg p(s) \vee \mathcal{X}(\bigvee_{(s, s_i) \in R} p(s_i))) \mid s \in S\}$$

If the KS is in state s in the current instant, in the next instant it is in one of the successors s_i of s .

$$CL_\top = \{\mathcal{G}(\neg p(s) \vee \alpha) \mid s \in S, \alpha \in AP_c, L_{\mathcal{A}}(s, \alpha) = \top\}$$

If the KS is in state s such that $L_{\mathcal{A}}(s, \alpha) = \top$, the atomic proposition α is true.

$$CL_\perp = \{\mathcal{G}(\neg p(s) \vee \neg \alpha) \mid s \in S, \alpha \in AP_c, L_{\mathcal{A}}(s, \alpha) = \perp\}.$$

If the KS is in state s such that $L_{\mathcal{A}}(s, \alpha) = \perp$, the atomic proposition α is false.

$$C_{REG} = \{\mathcal{G}(\neg p(s) \vee \neg p(s_i)) \mid s, s_i \in S \text{ and } s \neq s_i\}$$

The KS is in at most one state at any time.

5.2. Extracting the topological proofs

The procedure CTP_KS (Compute Topological Proofs) to compute x -TPs detailed in Algorithm 1 takes as input a PKS M , its optimistic/pessimistic approximation, i.e., here denoted generically as the KS \mathcal{A} , and an LTL formula $\phi_{neg} = F(\phi)$ —satisfied in \mathcal{A} (see Sect. 2). The three steps of the algorithm are described in the following.

5.2.1. SYS2LTL: Translating the KS and the property into an LTL formula

The KS $\mathcal{A} = \langle S, R, S_0, AP_c, L_{\mathcal{A}} \rangle$ and the LTL formula ϕ_{neg} are used to generate a set of clauses $C = C_{\mathcal{A}} \cup \{\phi_{neg}\}$. The clauses in $C_{\mathcal{A}}$ encode the KS. Specifically, $C_{\mathcal{A}} = C_{KS} \cup C_{REG}$, where $C_{KS} = \{c_i\} \cup C_N \cup CL_\top \cup CL_\perp$. The sets of clauses c_i , C_N , CL_\top and CL_\perp , C_{REG} are defined as specified in Table 3. The initial clause c_i specifies that the KS is initially in one of its initial states. The next state clauses in C_N specify that if in the current instant the KS is in state s , in the next instant it is in one of the successors states s_i of s . The labeling clauses in CL_\top specify that if the KS is in state s such that $L_{\mathcal{A}}(s, \alpha) = \top$, the atomic proposition α is true. The labeling clauses in CL_\perp specify that if the KS is in state s such that $L_{\mathcal{A}}(s, \alpha) = \perp$, the atomic proposition α is false. Finally, the regularity clauses in C_{REG} specify that the KS is in at most one state at any time. Note that the clauses in $C_{\mathcal{A}}$ are defined on the set of atomic propositions $AP_S = AP_c \cup \{p(s) \mid s \in S\}$, i.e., AP_S includes an additional atomic proposition $p(s)$ for each state s , which is true when the KS is in state s . The LTL formula ψ to be analyzed to check for conflicting clauses is defined as follows.

$$\psi = \bigwedge_{c \in C} c$$

Table 4. Rules to extract the TP-clauses from the conflicting LTL clauses.

LTL clause	TP clause	Type	LTL clause	TP clause	Type
$c_i = \bigvee_{s \in S_0} p(s)$	$\langle S_0 \rangle$	TPI	$\mathcal{G}(\neg p(s) \vee \neg \alpha)$	$\langle s, \alpha, \text{comp}(L(s, \alpha)) \rangle$	TPP
$\mathcal{G}(\neg p(s) \vee \mathcal{X}(\bigvee_{(s, s_i) \in R} p(s_i)))$	$\langle s, T \rangle$ and $T = \{s_i \mid (s, s_i) \in R\}$	TPT	$\mathcal{G}(\neg p(s) \vee \bar{\alpha})$	$\langle s, \alpha, \text{comp}(L(s, \alpha)) \rangle$	TPP
$\mathcal{G}(\neg p(s) \vee \alpha)$	$\langle s, \alpha, L(s, \alpha) \rangle$	TPP	$\mathcal{G}(\neg p(s) \vee \neg \bar{\alpha})$	$\langle s, \alpha, L(s, \alpha) \rangle$	TPP

The problem of model checking the KS \mathcal{A} and the LTL formula ϕ_{neg} is solved by checking the satisfiability of ψ (see Sect. 2). The number of clauses of ψ is the number of clauses in $\mathcal{C}_{\mathcal{A}}$, which is, in the worst case, $1 + |S| + |S| \times |AP_c| + |S| \times |S|$, plus one, that is the clause generated from the formula ϕ_{neg} .

5.2.2. GETTP: extracting the topological clauses

As detailed in Sect. 3, the GETTP component takes the subset C_{uc} of the conflicting clauses as input. As we will discuss in Sects. 5.3 and 5.4, $C_{uc} = \{C_{uc, \mathcal{A}} \cup \{\phi_{neg}\}\}$, where $C_{uc, \mathcal{A}} = C_{uc, KS} \cup C_{uc, REG}$ such that $C_{uc, KS} \subseteq C_{KS}$ and $C_{uc, REG} \subseteq C_{REG}$. Specifically, the set $C_{uc, \mathcal{A}}$ contains the clauses regarding the KS ($C_{uc, KS}$ and $C_{uc, REG}$) that made the formula ψ unsatisfiable. Since we are interested in clauses related to the KS that caused unsatisfiability, we extract the topological proof Ω , whose topological proof clauses are obtained from the clauses in $C_{uc, KS}$ as specified in Table 4. Since the set of atomic propositions of \mathcal{A} is $AP_c = AP \cup \bar{AP}$, in the table we use α for propositions in AP and $\bar{\alpha}$ for propositions in \bar{AP} . The table contains for each LTL clause its corresponding topological proof clause.

The elements in $C_{uc, REG}$ are not considered in the TP computation as, given an LTL clause $\mathcal{G}(\neg p(s) \vee \neg p(s_i))$, either state s or s_i is constrained by other TP-clauses that will be preserved in the model revisions.

Lemma 5.1 Let \mathcal{A} be a KS and let ϕ_{neg} be an LTL property. Let also ψ be the LTL formula computed in the step Sys2LTL of the algorithm, where $C = C_{\mathcal{A}} \cup \{\phi_{neg}\}$ and $C_{\mathcal{A}} = C_{REG} \cup C_{KS}$, and let ψ_{uc} be an unsatisfiable core, where $C_{uc} = C_{uc, \mathcal{A}} \cup \{\phi_{neg}\}$ and $C_{uc, \mathcal{A}} = C_{uc, REG} \cup C_{uc, KS}$. Then, if $\mathcal{G}(\neg p(s) \vee \neg p(s_i)) \in C_{uc, REG}$, either:

- (i) there exists an LTL clause in $C_{uc, KS}$ that constrains state s (or state s_i); or
- (ii) ψ'_{uc} , such that $C'_{uc} = C'_{uc, \mathcal{A}} \cup \{\phi_{neg}\}$ and $C'_{uc, \mathcal{A}} = C_{uc, \mathcal{A}} \setminus \{\mathcal{G}(\neg p(s) \vee \neg p(s_i))\}$, is an UC of ψ_{uc} .

Proof Sketch We indicate $\mathcal{G}(\neg p(s) \vee \neg p(s_i))$ as $\tau(s, s_i)$. Assume per absurdum that conditions (i) and (ii) are violated, i.e., no LTL clause in $C_{uc, KS}$ constrains state s or s_i (for condition (i)), and ψ'_{uc} is not an unsatisfiable core of ψ_{uc} (for condition (ii)). Since ψ'_{uc} is not an unsatisfiable core of ψ_{uc} , the LTL formula ψ'_{uc} where $C'_{uc} = C'_{uc, \mathcal{A}} \cup \{\phi_{neg}\}$ is satisfiable. Since C'_{uc} is satisfiable, C_{uc} must also be satisfiable since $C_{uc, \mathcal{A}} = C'_{uc, \mathcal{A}} \cup \{\tau(s, s_i)\}$. Indeed, since condition (i) is violated, it does not exist any LTL clause that constrains state s (or state s_i) and, in order to generate a contradiction, the added LTL clause must generate it using the LTL clauses obtained from the LTL property ϕ_{neg} . This is a contradiction and proves our lemma. \square

The ANALYZE procedure in Algorithm 2 obtains a TP (4, 6) for a PKS by first computing the related optimistic or pessimistic approximation (i.e., a KS) and then exploiting the computation of the TP for such KS. In the following we prove that our procedure is correct, i.e., given a PKS M and a property ϕ such that $[M \models \phi] = x$ it returns an x -topological proof for ϕ in M , that is every Ω -related PKS $M_{\Omega rel}$ is such that $[M_{\Omega rel} \models \phi] \geq x$.

Theorem 5.1 Let M be a PKS, let ϕ be an LTL property, and let $x \in \{\top, ?\}$ be an element such that $[M \models \phi] = x$. If the procedure ANALYZE, applied to the PKS M and the LTL property ϕ , returns a TP Ω for ϕ , this is an x -TP for ϕ in M .

Proof Sketch Assume that the ANALYZE procedure returns the value \top and a \top -TP for ϕ and the PKS $M = \langle S, R, S_0, AP, L \rangle$. We show that every Ω -related PKS $M_{\Omega rel}$ is such that $[M_{\Omega rel} \models \phi] \geq x$ (Definition 4.3). If ANALYZE returns the value \top , it must be that $M_{pes} \models^* \phi$ by Lines 5 and 7 of Algorithm 2. Furthermore, by Line 7, $\phi_{neg} = F(\phi)$ and $\mathcal{A} = M_{pes}$.

Let $CA = C_{\mathcal{A}} \cup \{\phi_{neg}\}$ be the clauses of the LTL formula associated with \mathcal{A} and ϕ_{neg} . Let us consider the clauses $CA_{uc} = C_{uc, \mathcal{A}} \cup \{\phi_{neg}\}$ of an UC CA , where $C_{uc, \mathcal{A}} = C_{uc, KS} \cup C_{uc, REG}$, $C_{uc, KS} \subseteq C_{KS}$ and $C_{uc, REG} \subseteq C_{REG}$.

Let $N = \langle S_N, R_N, S_{0, N}, AP_N, L_N \rangle$ be a PKS Ω -related to M . Let $CB = C_{\mathcal{B}} \cup \{\phi_{neg}\}$ be the clauses of the LTL formula associated with $\mathcal{B} = N_{pes}$ and ϕ_{neg} .

We show that $C_{uc,\mathcal{A}} \subseteq C_B$, i.e., the UC is also an UC for the LTL formula associated with the approximation \mathcal{B} of the PKS N .

As $C_{uc,\mathcal{A}} = C_{uc,KS} \cup C_{uc,REG}$ this is equivalent to check $(C_{uc,KS} \cup C_{uc,REG}) \subseteq C_B$. By Lemma 5.2 we can avoid considering $C_{uc,REG}$. By construction (see Line 2 of Algorithm 1) any clause $c \in C_{uc,KS}$ belongs to one rule among C_N , $CL_{pes,\top}$, $CL_{pes,\perp}$ or $c = c_i$:

- if $c = c_i$ then, by the rules in Table 4, there is a TPI-clause $\{S_0\} \in \Omega$. By Definition 4.2, $S_0 = S_{0,N}$. Thus, $c_i \in C_B$ since N is Ω -related to M .
- if $c \in C_N$ then, by the rules in Table 4, there is a TPT-clause $\langle s, T \rangle \in \Omega$ where $s \in S$ and $T \subseteq R$. By Definition 4.2, $T = \{s_i \in S_N \mid (s, s_i) \in R_N\}$. Thus, $c \in C_B$ since N is Ω -related to M .
- if $c \in CL_{\mathcal{A},\top}$ or $c \in CL_{\mathcal{A},\perp}$, by rules in Table 4, there is a TPP-clause $(s, \alpha, L(s, \alpha)) \in \Omega$ where $s \in S$ and $\alpha \in AP$. By Definition 4.2, $L_N(s, \alpha) = L(s, \alpha)$. Thus, $c \in C_B$ since N is Ω -related to M .

Since N is Ω -related to M , it has preserved the elements of Ω . Thus CA is also an UC of CB . It follows that $[N \models \phi] = \top$.

The proof from the case in which ANALYZE procedure returns the value ? and a ?-TP can be derived from the first case. \square

5.3. Computing the conflicting clauses in ψ with PLTL-MUP

We now describe the procedure used by TOrPEDO-MUP to extract the conflicting clauses of ψ . Recall that the set of clauses of the LTL formula ψ encodes the KS \mathcal{A} and the property ϕ_{neg} . As mentioned in Sect. 3.1, none of the behaviors of \mathcal{A} satisfies the property ϕ_{neg} since ϕ_{neg} encodes the behaviors that violate ϕ which is satisfied by \mathcal{A} . Therefore, ψ is unsatisfiable. The unsatisfiable core (UC) contains the set of conflicting clauses that lead to the contradiction. TOrPEDO-MUP uses PLTL-MUP [SGT13] to extract the unsatisfiable core of ψ . PLTL-MUP exploits Binary Decision Diagrams (BDDs) and a BDD-based theorem prover for LTL to compute the UC.

The function $\psi_{uc} = \text{GETUC}(\psi)$ returns an unsatisfiable core $\psi_{uc} = \bigwedge_{c \in C_{uc}} c$ of ψ . Specifically, it returns a subset of clauses $C_{uc} = C_{uc,\mathcal{A}} \cup \{\psi\}$, where $C_{uc,\mathcal{A}} = C_{uc,KS} \cup C_{uc,REG}$ such that $C_{uc,KS} \subseteq C_{KS}$ and $C_{uc,REG} \subseteq C_{REG}$.

Lemma 5.2 Let \mathcal{A} be a KS and let ϕ_{neg} be an LTL property. Let $C = \{C_{\mathcal{A}} \cup \{\phi_{neg}\}\}$ be the set of clauses generated from the KS \mathcal{A} and the LTL formula ϕ_{neg} , and ψ be the LTL formula computed in the step SYS2LTL of the algorithm. Then, any unsatisfiable core ψ_{uc} of ψ is made by a subset of clauses $C_{uc} = \{C_{uc,\mathcal{A}} \cup \{\phi_{neg}\}\}$ such that $C_{uc,\mathcal{A}} \subseteq C_{\mathcal{A}}$.

Proof Sketch As the property ϕ_{neg} is satisfied by M , the LTL formula ψ is unsatisfiable (see Sect. 2). Since $C_{\mathcal{A}}$ encodes a KS, $\bigwedge_{c \in C_{\mathcal{A}}} c$ is satisfiable. If ϕ_{neg} is satisfiable (i.e., it is not vacuously true or false), then the unsatisfiability is caused by the contradiction of some of the clauses in $C_{\mathcal{A}}$ and the property ϕ_{neg} , and as a consequence ϕ_{neg} must be a part of the UC, $C_{uc,\mathcal{A}}$ is not empty, and $C_{uc,\mathcal{A}} \subseteq C_{\mathcal{A}}$. \square

After detecting the set C_{uc} of conflicting LTL clauses, the procedure described in Sect. 5.2.2 is used to generate the TP.

5.4. SMT-based extraction of topological proofs

This section describes the procedure to translate LTL clauses into propositional logic (PL) clauses (Sect. 5.4.1). Then, it presents the procedure used to detect conflicts among the propositional logic clauses (Sect. 5.4.2), and describes how the conflicting LTL clauses are extracted from the conflicting PL logic clauses (Sect. 5.4.3). Finally, it discusses the correctness of our procedure (Sect. 5.4.4).

5.4.1. LTL2PL: from linear time temporal logic to propositional logic

Recall that the LTL formula ψ —to be analyzed to check for conflicting clauses in C —is defined as follows.

$$\psi = \bigwedge_{c \in C} c$$

For translating the LTL formula ψ into PL we use the technique proposed by Schuppan et al. [SLJ⁺06], also used in more recent works (e.g., [BKR15]). The technique is based on two observations:

1. an LTL formula ψ is satisfiable if there exists an infinite path that satisfies ψ ; and
2. the infinite paths that satisfy LTL formulae can be represented as $\pi = s_0, s_1, \dots, s_{r-1}, (s_r, s_{r+1}, \dots, s_k)^\omega$, where $(s_r, s_{r+1}, \dots, s_k)^\omega$ indicates that, after state s_k is left, the path restarts from s_r . The infinite path is made by two parts: the *prefix*, i.e., s_0, s_1, \dots, s_{r-1} , and the ultimately *periodic part* s_r, s_{r+1}, \dots, s_k .

Building on these two observations, the idea is to check the satisfiability of ψ by: i) generating a propositional formula φ encoding all the possible ultimately periodic paths (up to length k) that satisfy ψ and ii) checking for the satisfiability of φ . If φ is satisfiable, there exists an ultimately periodic path that satisfies ψ . Therefore, the LTL formula ψ is satisfiable. If φ is unsatisfiable and k is “big enough”, then ψ is unsatisfiable. In Sect. 5.4.4 we will discuss the implications of the k selection on the correctness of our procedure.

The propositional formula φ is made by two parts, one encoding the *infinite (ultimately periodic) paths*, and one encoding the *semantics of the formula ψ* . These two parts are described in the following.

Encoding the Infinite (ultimately periodic) Paths. The encoding introduces the following variables and PL formulae, in which the symbols Ξ_i , with $i \in 0, 1, \dots$ are used to identify each PL formula.

- A set of *loop selector variables* (l_0, l_1, \dots, l_k). These are new fresh Boolean variables such that l_i is true if the ultimately periodic part starts at position i . To enforce the fact that the ultimately periodic part starts from one state, only one of the loop selector variables must be true. This is enforced by the following formula:

$$\Xi_0 \equiv \bigwedge_{i \in \{0, \dots, k\}} \left(l_i \Leftrightarrow \bigwedge_{j \neq i, j \in \{0, \dots, k\}} \neg l_j \right) \quad (3)$$

For each index i , the formula specifies that if l_i is true, all the other loop selector variables are false. Furthermore, we force one of the loop selector variables to be true by adding the following formula:

$$\Xi_1 \equiv l_0 \vee l_1 \vee \dots \vee l_k \quad (4)$$

- A set of *in loop variables* ($InLoop_0, InLoop_1, \dots, InLoop_k$). These are new fresh Boolean variables such that $InLoop_i$ is true if the state s_i is in the ultimately periodic part. To enforce the correspondence between the loop selector variables and the in loop variables, we add the following propositional formula.

$$\Xi_2 \equiv \bigwedge_{i \in \{0, \dots, k\}} (InLoop_i \Leftrightarrow (InLoop_{i-1} \vee l_i)) \quad (5)$$

This formula imposes that $InLoop_i$ is in the loop only if the ultimately periodic part starts at position i or it starts in a previous position, i.e., $InLoop_{i-1}$ is true.

- The variable *LoopExists*. It is true if and only if a loop has been found. To force this relation we add the following formula.

$$\Xi_3 \equiv LoopExists \Leftrightarrow (l_0 \vee l_1 \vee \dots \vee l_k) \quad (6)$$

Table 5. Propositional formulae added to the encoding to capture the LTL semantics.

ψ_j	PL
p	$\bigwedge_{i \in \{0, \dots, k\}} \lceil p \rceil_i$
$\neg \psi$	$\bigwedge_{i \in \{0, \dots, k\}} \lceil \neg \psi \rceil_i \Leftrightarrow \neg \lceil \psi \rceil_i$
$\mathcal{X} \psi$	$\bigwedge_{i \in \{0, \dots, k\}} \lceil \mathcal{X} \psi \rceil_i \Leftrightarrow \lceil \psi \rceil_{i+1}$
$\psi_1 \wedge \psi_2$	$\bigwedge_{i \in \{0, \dots, k\}} \lceil \psi_1 \wedge \psi_2 \rceil_i \Leftrightarrow \lceil \psi_1 \rceil_i \wedge \lceil \psi_2 \rceil_i$
$\psi_1 \mathcal{U} \psi_2$	$\bigwedge_{i \in \{0, \dots, k\}} \lceil \psi_1 \mathcal{U} \psi_2 \rceil_i \Leftrightarrow (\lceil \psi_2 \rceil_i \vee (\lceil \psi_1 \rceil_i \wedge \lceil \psi_1 \mathcal{U} \psi_2 \rceil_i)) \wedge \bigwedge_{i \in \{0, \dots, k\}} (LoopExists \Rightarrow (\lceil \psi_1 \mathcal{U} \psi_2 \rceil_i \Rightarrow \lceil \mathcal{F} \psi_2 \rceil_i)) \wedge \bigwedge_{i \in \{0, \dots, k\}} (\lceil \mathcal{F} \psi_2 \rceil_i \Leftrightarrow (\lceil \mathcal{F} \psi_2 \rceil_{i-1} \vee (InLoop_i \wedge \lceil \mathcal{F} \psi_2 \rceil_i))) \wedge \lceil \mathcal{F} \psi_2 \rceil_0 = \perp$

Encoding the Semantics of the Formula ψ . To force the ultimately periodic path to satisfy the LTL formula, we need to encode how the formula is satisfied on the different states of the path. This should be specified according to the semantics of the LTL formula. In the following, we describe how each LTL clause $c \in C$ is translated into a PL formula φ_c . Let $\psi_0, \psi_1, \dots, \psi_h$ be the subformulae of a clause c of the LTL formula ψ . To represent the LTL clause c in PL, we introduce a Boolean variable $\lceil \psi_j \rceil_i$ for each index i in $0, \dots, k+1$ and subformula ψ_j of c . Each variable $\lceil \psi_j \rceil_i$ represents the value of a subformula ψ_j in the position i of the path. We consider indexes from $0, \dots, k+1$ since, to represent the ultimately periodic part of the path, we need to enforce that LTL subformulae satisfied by the state s_{k+1} correspond to the LTL subformulae satisfied by the state s_r , i.e., in the first state of the ultimately periodic part of the path. We use the symbols ξ_i , with $i \in 0, 1, \dots$ to identify each one of the PL formulae. To enforce that the subformulae that hold in s_{k+1} are the same that hold in state s_r , we add the following formula:

$$\xi_0 \equiv \bigwedge_{i \in \{0, \dots, k\}} \left(\bigwedge_{j \in \{0, \dots, h\}} (l_i \Rightarrow (\lceil \psi_j \rceil_i \Leftrightarrow \lceil \psi_j \rceil_{k+1})) \right) \quad (7)$$

Then, we enforce the LTL semantics by using the standard fix-point encoding specified in Table 5. For every subformula $\psi_0, \psi_1, \dots, \psi_h$ of c , a PL formula (i.e., $\xi_1, \xi_2, \dots, \xi_{h+1}$) is created depending on the type of the constructs used in the subformula (see the PL labelled column of Table 5). The encoding of $p, \neg \psi, \mathcal{X} \psi$, and $\psi_1 \wedge \psi_2$ directly follows from the LTL semantics. For $\psi_1 \mathcal{U} \psi_2$, the encoding specifies that $\lceil \psi_1 \mathcal{U} \psi_2 \rceil_i \Leftrightarrow (\lceil \psi_2 \rceil_i \vee (\lceil \psi_1 \rceil_i \wedge \lceil \psi_1 \mathcal{U} \psi_2 \rceil_i))$ is true for every index i according to the semantics of the \mathcal{U} temporal operator. However, this formula is vacuously satisfied when $\lceil \psi_1 \rceil_i \wedge \lceil \psi_1 \mathcal{U} \psi_2 \rceil_i$ is true for every index i that belongs to the ultimately periodic part, but ψ_2 is never satisfied. To avoid this case, we need to ensure that ψ_2 eventually occurs. This is achieved by

1. the formula $LoopExists \Rightarrow (\lceil \psi_1 \mathcal{U} \psi_2 \rceil_k \Rightarrow \lceil \mathcal{F} \psi_2 \rceil_k)$, specifying that whenever an ultimately periodic path is present ($LoopExists$ is true), if $\psi_1 \mathcal{U} \psi_2$ is satisfied in position k , $\mathcal{F} \psi_2$ is also satisfied in position k ;
2. the formula $\bigwedge_{i \in \{0, \dots, k\}} (\lceil \mathcal{F} \psi_2 \rceil_i \Leftrightarrow (\lceil \mathcal{F} \psi_2 \rceil_{i-1} \vee (InLoop_i \wedge \lceil \mathcal{F} \psi_2 \rceil_i)))$, specifying that $\mathcal{F} \psi_2$ holds in a position i of the loop (and therefore in position k) if $\mathcal{F} \psi_2$ holds in a previous position of the loop; and
3. the formula $\lceil \mathcal{F} \psi_2 \rceil_0 = \perp$ forces $\mathcal{F} \psi_2$ to be initially false.

The final PL formula φ_c —that represents the LTL clause c —is obtained by combining the formula with the conjunction Boolean operator, i.e., φ_c is defined as follows:

$$\varphi_c \equiv \xi_0 \wedge \xi_1 \wedge \dots \wedge \xi_{h+1} \quad (8)$$

Then, the satisfiability of ψ can be verified by checking the satisfiability of:

$$\varphi = \Xi_0 \wedge \Xi_1 \wedge \Xi_2 \wedge \Xi_3 \wedge \bigwedge_{c \in C} \varphi_c$$

Note that, since our goal is to reduce the computational cost required to compute topological proofs, we implemented the LTL2PL translation by relying on an encoding based on Bit-Vectors [BKR15]. According to some recent result, this encoding provides significant benefits with respect to existing tools [BKR15, PKRB20].

5.4.2. GETUC: computing the unsatisfiable core of a PL formula

To check whether the formula φ is satisfiable and to extract its unsatisfiable core, we employ the Z3 Theorem Prover [DMB08]; we selected this solver as it extracts unsatisfiable cores and it is an industry-strength tool,

also awarded by ETAPS (Test of Time Award) [sig20] and ACM SIGPLAN (Programming Languages Software Award) [eta20].

Given a formula and a set of clauses, a.k.a. assumptions, Z3 checks whether the formula is satisfiable and identifies the clauses that are in contradiction. Specifically, given the PL formula φ and the set of PL clauses $\varphi_1, \varphi_2, \dots, \varphi_n$, each one generated from an LTL clause $c_i \in C$, $Z3.unsat_core$ checks whether φ is satisfiable, and—in case it is not—it returns a subset Π of the set of the PL clauses $\{\varphi_1, \varphi_2, \dots, \varphi_n\}$ that lead to the contradiction.

$$\Pi = Z3.unsat_core(\varphi, \varphi_1, \varphi_2, \dots, \varphi_n) \quad (9)$$

For example, if φ is unsatisfiable, Π might contain the PL clauses φ_2 and φ_3 generated from the LTL clauses c_2 and c_3 that are leading to the contradiction, i.e., $\Pi = \{\varphi_2, \varphi_3\}$.

5.4.3. GETTP: mapping the conflicting propositional clauses to LTL

The function GETTP aims at identifying—from the set of the conflicting PL clauses Π —the corresponding LTL clauses. Recall that, as discussed in Sect. 5.4.1, each clause within the set of PL clauses $\{\varphi_1, \varphi_2, \dots, \varphi_n\}$ processed by Z3 is obtained from an LTL clause $c \in C$. Therefore, given the set of the conflicting PL clauses Π , our algorithm computes the set C_{uc} of conflicting LTL clauses by adding to C_{uc} —for each PL formulae φ_c in Π —the LTL clause c from which it was generated.

After detecting the set C_{uc} of conflicting LTL clauses, the procedure described in Sect. 5.2.2 is used to generate the TP.

5.4.4. Correctness

The unsatisfiable core Π contains the set of PL clauses of $\{\varphi_1, \varphi_2, \dots, \varphi_n\}$ that lead to a contradiction. As specified in Sect. 5.4.1, each PL clause in $\{\varphi_1, \varphi_2, \dots, \varphi_n\}$ corresponds to an LTL clause $\{c_1, c_2, \dots, c_n\}$ in C . The algorithm extracts from Π the subset C_{uc} of C containing the LTL clauses of C corresponding to the PL clauses leading to the contradiction. The algorithm is *correct* if the LTL clauses in C_{uc} are contradicting.

The bounded encoding presented in Sect. 5.4.1 finds ultimately periodic paths (up to length k , a.k.a. bound) that satisfy the formula ψ , but cannot prove their absence. To prove that such paths do not exist, it must be shown that a path that satisfies the LTL formula ψ cannot be longer than a certain bound, a.k.a. completeness threshold. Recall that, as for classical bounded model checking (BMC), the LTL formula ψ encodes the KS \mathcal{A} and the LTL formula ϕ_{neg} to be checked. Therefore, the completeness threshold can be identified by reusing existing approaches from the literature (see for example [SLJ⁺06, CKOS04, KOS⁺11]). Therefore, if the value of k is higher than the completeness threshold, there is formal guarantee that the LTL clauses in C_{uc} are contradicting. Vice versa, if k is lower than the completeness threshold, the clauses in C_{uc} may not be contradicting.

In practice, designers can initially choose a value for k that is reasonably large for the considered PKS and property. Then, they can increase or decrease the value of k depending on (a) the efficiency of the ANALYSIS component, and (b) how relevant is the soundness of the computed topological proofs for their application domain.

5.5. Re-check

Let $M = \langle S, R, S_0, AP, L \rangle$ be a PKS. The RE-CHECK algorithm verifies whether a revision $M_{\Omega rv}$ of M is an Ω -revision. Let Ω be an x -TP (10) for ϕ in M , and let $M_{\Omega rv} = \langle S_{\Omega rv}, R_{\Omega rv}, S_{\Omega rv,0}, AP_{\Omega rv}, L_{\Omega rv} \rangle$ be a revision of M (8). The RE-CHECK algorithm returns **true** if and only if the following holds:

- $AP \subseteq AP_{\Omega rv}$;
- for every TPP-clause $\langle s, \alpha, v \rangle \in \Omega$, $s \in S_{\Omega rv}$, $v = L_{\Omega rv}(s, \alpha)$;
- for every TPT-clause $\langle s, T \rangle \in \Omega$, $s \in S_{\Omega rv}$, $T \subseteq S_{\Omega rv}$, $T = \{s_i \in S_{\Omega rv} \mid (s, s_i) \in R_{\Omega rv}\}$;
- for every TPI-clause $\langle S_0 \rangle \in \Omega$, $S_0 = S_{\Omega rv,0}$.

These conditions can be verified by a simple syntactic check on the PKS. For example, considering the x -TP for the property ϕ_2 presented in Table 2 of our vacuum-cleaner example, the RE-CHECK algorithm returns a **true** value for the revision $M_{\Omega rv}$ in Fig. 1b of the model M in Fig. 1a.

Lemma 5.3 Let $M = \langle S, R, S_0, AP, L \rangle$ and $M' = \langle S', R', S'_0, AP', L' \rangle$ be two PKSs and let Ω be an x -TP. The RE-CHECK algorithm returns `true` if and only if M' is Ω -related to M .

Proof Sketch For M' to be Ω -related to M , the conditions of Definition 4.2 should hold. Each one of these conditions is a condition of the RE-CHECK algorithm. Thus, if M' is Ω -related to M , the RE-CHECK returns `true`. Conversely, if RE-CHECK returns `true`, each condition of the algorithm is satisfied and, since each of these conditions corresponds to a condition of Definition 4.2, M' is Ω -related to M . \square

The reported Lemma allows us to prove the following Theorem.

Theorem 5.2 Let M be a PKS, let ϕ be a property, let Ω be an x -TP for ϕ in M where $x \in \{\top, ?\}$, and let M_{rv} be a revision of M . The RE-CHECK algorithm returns `true` if and only if M_{rv} is an Ω_x -revision of M .

Proof Sketch By applying Lemma 5.3, the RE-CHECK algorithm returns `true` if and only if M_{rv} is Ω -related to M . By Definition 4.4, since Ω is an x -TP, the RE-CHECK algorithm returns `true` if and only if M_{rv} is an Ω_x -revision of M . \square

The ANALYSIS and RE-CHECK algorithms assume that the three-valued LTL semantics is considered. An alternative semantics, called thorough LTL semantics [BG00], has been introduced to provide an evaluation of formulae that better reflects the natural intuition. It has been demonstrated that the two semantics coincide in the case of self-minimizing LTL formulae [GH05]. In this case, our results are correct also w.r.t. the thorough semantics. Note that, as shown in [GH05], most practically useful LTL formulae are self-minimizing.

6. Evaluation

We implemented TOrPEDO as a Scala stand alone application and made it available online [Tor20]. We implemented TOrPEDO-MUP and TOrPEDO-SMT by using NuSMV (v2.6.0) and Z3 (v4.4 build 2 rev 1). We evaluated how the ANALYSIS helps in creating models revisions, how frequently running the RE-CHECK algorithm allows the user to avoid the re-execution of the ANALYSIS algorithm from scratch. We also evaluated the efficiency of TOrPEDO and how it supports the development of (small) models. Specifically, we considered the following research questions:

RQ1: How does the size of the proofs generated by the ANALYSIS algorithm of TOrPEDO-MUP compares to the size of the models being analyzed? (Sect. 6.1)

RQ2: How frequently running the RE-CHECK algorithm of TOrPEDO-MUP allows to avoid the re-execution of the ANALYSIS algorithm from scratch? (Sect. 6.2)

RQ3: How efficient is TOrPEDO in analyzing models and how does TOrPEDO-SMT compare to TOrPEDO-MUP? (Sect. 6.3)

RQ4: How useful is TOrPEDO-SMT in supporting the designers in the model design on an example in the genomic domain? (Sect. 6.4)

We used TOrPEDO-MUP for **RQ1** and **RQ2** since the analysis of the models considered for answering these questions did not require the most efficient version of our tool. We used TOrPEDO-SMT for **RQ3** and **RQ4**. We run our experiments on a machine with processor Intel Core i5 3.2GHz and 32GB of memory.

6.1. Analysis support—RQ1

To answer RQ1, we checked how the size of the proofs generated by the ANALYSIS algorithm compares to the size of the models being analyzed. The topological proofs represent constraints that, if satisfied, ensure that the property is not violated (or possibly violated). As discussed in Sect. 4, to ensure that the property is not violated, the designers should not modify the parts of the model constrained by the proofs while creating model revisions. Therefore, the smaller the topological proofs are, the more useful they are, since more elements can be changed during the model revisions. In addition, smaller proofs allow for easier inspection, and this makes it easier to create revisions.

Table 6. Properties considered in the evaluation

ϕ_1 :	$\mathcal{G}(\neg OFFHOOK) \vee (\neg OFFHOOK \mathcal{U} CONNECTED)$
ϕ_2 :	$\neg OFFHOOK \mathcal{W} (\neg OFFHOOK \wedge CONNECTED)$
ϕ_3 :	$\mathcal{G}(CONNECTED \rightarrow ACTIVE)$
ϕ_4 :	$\mathcal{G}(OFFHOOK \wedge ACTIVE \wedge \neg CONNECTED \rightarrow \mathcal{X}(ACTIVE))$
ϕ_5 :	$\mathcal{G}(CONNECTED \rightarrow \mathcal{X}(ACTIVE))$
ψ_1 :	$\mathcal{G}(CONNECTED \rightarrow ACTIVE)$
ψ_2 :	$\mathcal{G}(CONNECTED \rightarrow \mathcal{X}(ACTIVE))$
ψ_3 :	$\mathcal{G}(CONNECTED) \vee (CONNECTED \mathcal{U} \neg OFFHOOK)$
ψ_4 :	$\neg CONNECTED \mathcal{W} (\neg CONNECTED \wedge OFFHOOK)$
ψ_5 :	$\mathcal{G}(CALLEE_SEL \rightarrow OFFHOOK)$
η_1 :	$\mathcal{G}((OFFHOOK \wedge CONNECTED) \rightarrow \mathcal{X}(OFFHOOK \vee \neg CONNECTED))$
η_2 :	$\mathcal{G}(CONNECTED) \vee (CONNECTED \mathcal{W} \neg OFFHOOK)$
η_3 :	$\neg CONNECTED \mathcal{W} (\neg CONNECTED \wedge OFFHOOK)$
η_4 :	$\mathcal{G}(CALLEE_FREE \vee LINE_SEL)$
η_5 :	$\mathcal{G}(\mathcal{X}(OFFHOOK) \wedge \neg CONNECTED)$

Table 7. Cardinalities $|S|$, $|R|$, $|AP|$, $|?|$, and $|M|$ are those of the evaluated model M . $|\Omega_p|_x$ is the size of proof Ω_p for a property p ; x indicates if Ω_p is a T-TP or a ?-TP. The column $|\Omega_p|_x$ also reports percentage of the ratio between the size of the proof and the size of the model.

Model	$ S $	$ R $	$ AP $	$? $	$ M $	$ \Omega_{\phi_1} $	$ \Omega_{\phi_2} $	$ \Omega_{\phi_3} $	$ \Omega_{\phi_4} $	$ \Omega_{\phi_5} $
<i>callee-1</i>	5	15	3	7	31	$7_?$ (22%)	$9_?$ (29%)	$21_?$ (68%)	$23_?$ (74%)	$23_?$ (74%)
<i>callee-2</i>	5	15	3	4	31	$7_?$ (22%)	$9_?$ (29%)	$21_?$ (68%)	$22_?$ (71%)	×
<i>callee-3</i>	5	15	3	2	31	$7_?$ (22%)	$9_?$ (29%)	$21_?$ (68%)	$23_?$ (74%)	×
<i>callee-4</i>	5	15	3	0	31	×	×	$23_?$ (74%)	$21_?$ (68%)	×
Model	$ S $	$ R $	$ AP $	$? $	$ M $	$ \Omega_{\psi_1} $	$ \Omega_{\psi_2} $	$ \Omega_{\psi_3} $	$ \Omega_{\psi_4} $	$ \Omega_{\psi_5} $
<i>caller-1</i>	6	21	5	4	52	$28_?$ (54%)	×	$2_?$ (4%)	$9_?$ (17%)	$28_?$ (54%)
<i>caller-2</i>	7	22	5	4	58	$30_?$ (52%)	×	$2_?$ (4%)	$9_?$ (16%)	$30_?$ (52%)
<i>caller-3</i>	6	19	5	1	50	$26_?$ (52%)	$28_?$ (56%)	$2_?$ (4%)	$11_?$ (22%)	$26_?$ (52%)
<i>caller-4</i>	6	21	5	0	52	$28_?$ (54%)	×	$2_?$ (4%)	$9_?$ (17%)	$28_?$ (54%)
Model	$ S $	$ R $	$ AP $	$? $	$ M $	$ \Omega_{\eta_1} $	$ \Omega_{\eta_2} $	$ \Omega_{\eta_3} $	$ \Omega_{\eta_4} $	$ \Omega_{\eta_5} $
<i>caller-callee-1</i>	6	30	6	30	61	$37_?$ (61%)	$2_?$ (3%)	$15_?$ (25%)	$37_?$ (61%)	×
<i>caller-callee-2</i>	7	35	6	36	78	$43_?$ (55%)	$2_?$ (3%)	$18_?$ (23%)	$43_?$ (55%)	×
<i>caller-callee-3</i>	7	45	6	38	88	$53_?$ (60%)	$2_?$ (3%)	$53_?$ (60%)	$53_?$ (60%)	$53_?$ (60%)
<i>caller-callee-4</i>	6	12	4	0	42	×	×	×	$19_?$ (45%)	×

The goal of this research question is to assess how useful the computed topological proofs are. The usefulness is evaluated by comparing the size of the topological proofs to the size of the models being analyzed.

Dataset. We considered a set of 3 examples (*callee*, *caller* and *caller-callee*) proposed in the literature and used to evaluate χ Chek [ECD⁺03]. For each of these examples, we consider the PKS representing the initial model, indicated using the name of the model followed by the index 1, and three model revisions, each indicated using the name of the model following an incremental index. For example, *callee-1* indicates the initial PKS model for the *callee* example, while *callee-2*, *callee-3*, and *callee-4* are three consecutive revisions of *callee-1*. Therefore, in our evaluation, we considered 12 PKS in total. Table 7 reports the cardinalities $|S|$, $|R|$ and $|AP|$ of the sets of states, transitions, and atomic propositions of each considered PKS. The column with label $|?|$ contains the number of combinations made by a state s and an atomic propositions α such that $L(s, \alpha) = ?$. The number of states, transitions, atomic propositions, and combinations of a state s and an atomic proposition α such that $L(s, \alpha) = ?$ is different across the different examples. This shows that our examples are sufficiently diverse to enable us assessing how the size of the proofs generated by the ANALYSIS algorithm of TOrPEDO-MUP compares to the size of the models.

We considered five properties for each example (see Table 6). These properties were inspired by the original properties and based on the LTL property patterns [DAC99].³ These properties are sufficiently diverse to support our experiments, since they use all the LTL operators.⁴ Therefore, they enable us to assess the topological proofs computed starting from different LTL operators.

Methodology. We run the ANALYSIS component of TOrPEDO. We considered each of the 60 model-property combinations of our dataset. For each model-property combination, we executed the ANALYSIS component of TOrPEDO and we recorded its output (see Fig. 2).

³ The original properties used in the examples were specified in Computation Tree Logic (CTL), which is currently not supported by TOrPEDO.

⁴ The weak until operator (\mathcal{W}) is rewritten using the until operator (\mathcal{U}) following the standard LTL rewriting rules.

Table 8. Results returned by the RE-CHECK component for the models and the properties of our benchmark.

Model	ϕ_1	ϕ_2	ϕ_3	ϕ_4	ϕ_5	Model	ψ_1	ψ_2	ψ_3	ψ_4	ψ_5	Model	η_1	η_2	η_3	η_4	η_5
<i>callee-1</i>	-	-	-	-	-	<i>caller-1</i>	-	-	-	-	-	<i>caller-callee-1</i>	-	-	-	-	-
<i>callee-2</i>	✓	✓	✓	✓	✗	<i>caller-2</i>	✓	-	✓	✓	✓	<i>caller-callee-2</i>	✓	✓	✓	✓	-
<i>callee-3</i>	✓	✓	✓	✓	-	<i>caller-3</i>	✓	-	✓	✓	✓	<i>caller-callee-3</i>	✓	✓	✓	✓	-
<i>callee-4</i>	✗	✗	✓	✓	-	<i>caller-4</i>	✓	✗	✓	✓	✓	<i>caller-callee-4</i>	✗	✗	✗	✓	✗

For the cases in which TOrPEDO produced a topological proof, we also computed the size of the proof, as defined in Sect. 4. We compared the size of the PKS model and the size of the proofs returned by TOrPEDO.

Results. Table 7 summarizes the obtained results. We show the total size $|M|$ of the model, and the size $|\Omega_p|$ of the proofs. The column $|\Omega_p|_x$ also reports within rounded brackets the percentage of the ratio between the size of the proof and the size of the model. Cells labeled with the symbol \times indicate that a property was not satisfied in that model and thus a proof was not produced by the ANALYSIS algorithm. The numbers indicating the size of the proofs are tagged with a subscript that indicates whether the property is satisfied ($x = \top$) or possibly satisfied ($x = ?$). Proofs are $\approx 60\%$ smaller than their respective initial models. Thus, we conclude that the proofs are significantly more concise than the original model, thus, enabling a flexible design.

The answer to **RQ1** is that, on the considered models, TOrPEDO provides proofs that are $\approx 60\%$ smaller than their respective initial models.

6.2. Re-check support—RQ2

To answer RQ2, we checked how the results output by the RE-CHECK algorithm were useful in producing PKSs revisions.

Dataset. We considered the same dataset used for RQ1. We assumed that, for each example, the designer produced revisions following the order specified in Table 8 (column Model), that is consecutive revisions are identified using consecutive indexes labeling the name of the same example. For example, the model *callee-3* is a revision of the model *callee-2* which, in turn, is a revision of the model *callee-1*. Since the design of *callee-2* precedes the design of *callee-3*, we say that *callee-2* is the previous model of *callee-3*. The other columns contain the different properties that have been analyzed for each category.

Methodology. We run the RE-CHECK component of TOrPEDO on the models and the properties of our dataset. We recorded the output of the RE-CHECK component. We assessed in how many cases the RE-CHECK component allowed designers to avoid re-running the ANALYSIS.

Results. Table 8 reports our results. A cell contains ✓ if the RE-CHECK was passed by the considered revised model, i.e., a true value was returned by the RE-CHECK algorithm, ✗ otherwise. For example, the ✓ symbol associated by the model *callee-2* and the property ϕ_1 indicates that the RE-CHECK component confirmed that the revision *callee-2* of *callee-1* is an Ω -revision of *callee-1* by considering the topological proof Ω for ϕ_1 in *callee-1*. The dash symbol - is used when the model of the corresponding line is not a revision (i.e., the first model of each category) or when the observed property was false in the previous model, i.e., an x -TP for the property was not produced. For example, the model *caller-1* is not a revision, since it is the first model proposed for the *caller* example. Differently, the model *caller-2* and the property ψ_2 are associated with the symbol - since, as specified in Table 7, the ANALYSIS component did not produce any topological proof of the property ψ_2 in the model *caller-1*. Therefore, the RE-CHECK could not be executed. We inspected the results produced by the RE-CHECK algorithm to evaluate their benefit in verifying if revisions were violating the proofs. Table 8 shows that, in $\approx 21\%$ (number of cells labeled with the ✗ as a percentage of the number of cells labeled with symbols ✗ or ✓) of the cases, the TOrPEDO RE-CHECK notified the designer that the proposed revision violated some of the clauses contained in the Ω -proof, while in $\approx 79\%$ (number of cells labeled with the ✓ as a percentage of the number of cells labeled with symbols ✗ or ✓) the RE-CHECK allowed designers to avoid re-running the ANALYSIS (and thus the model checker).

The answer to **RQ2** is that, on the considered models, in $\approx 79\%$ the RE-CHECK component allowed designers to avoid re-running the ANALYSIS (and thus the model checker).

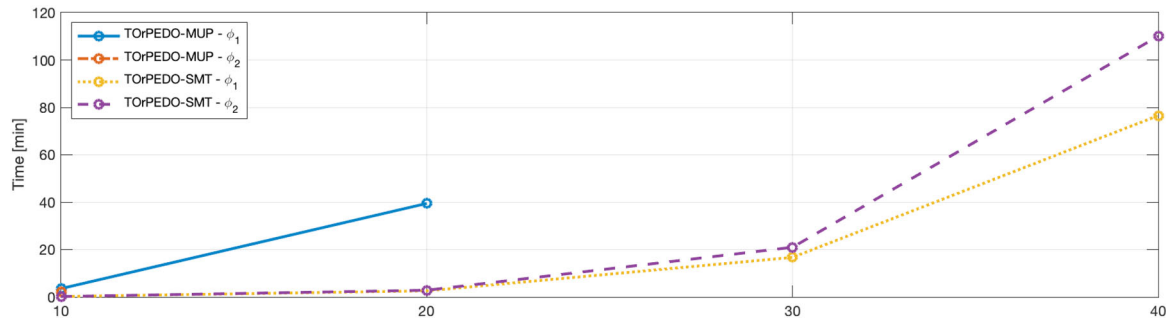


Fig. 4. Comparison of the efficiency of T0rPEDO-MUP and T0rPEDO-SMT. For the property ϕ_2 , T0rPEDO-MUP provided a result only for the model with 10 states in 2.1m

6.3. Efficiency—RQ3

To evaluate the efficiency of T0rPEDO we could not use the models of RQ1 and RQ2. Indeed, executing the ANALYSIS and RE-CHECK phases of T0rPEDO requires less than a minute for each property of each model. Therefore, analyzing the performances on these models does not provide significant practical results. For this reason, to evaluate the efficiency of T0rPEDO we analyzed a set of randomly generated models with increasing size. The ANALYSIS phase of T0rPEDO combines three-valued model checking and UCs computation. Three-valued model checking is as expensive as classical model checking [BG99], i.e., it is linear in the size of the model and exponential in the size of the property. UCs computation is FSPACE complete [SHH12]. The RE-CHECK phase performs a simple syntactic check; the complexity of the RE-CHECK algorithm is linear in the size of the model. Therefore, we only analyze the efficiency of the ANALYSIS phase of T0rPEDO since the RE-CHECK requires negligible time compared to the ANALYSIS.

Dataset. We generated a set of random models with an increasing number of states (i.e., 10, 20, 30, and 40). The random models are generated from the grade crossing semaphore (GC) example ([BMS⁺17]) starting from the GC model, and by iteratively duplicating the GC model and connecting the duplicated model with the initial model with randomly generated transitions. We considered two properties ϕ_1 and ϕ_2 that are respectively satisfied and possibly satisfied on the GC model and on the randomly generated models. Property ϕ_1 specifies that *red* lights up infinitely often ($\square \diamond red$). Property ϕ_2 states that *green* lights up infinitely often ($\square \diamond green$).

Methodology. We run the ANALYSIS component of T0rPEDO-MUP and T0rPEDO-SMT tool by considering all the eight combinations made by a randomly generated model and a property. Note that, in our experiments we considered an extended version of PLTL-MUP, namely Hybrid, that improves the PLTL-MUP performances by combining it with TRP++UC [SGT13]. We set two hours as timeout for each run. For T0rPEDO-SMT, for each model and property, we set 86 as a value for the completeness bound k . We selected this value since it ensures the correctness of the result, i.e., we set its value by considering to the size of the recurrence diameter (the longest initialised loop-free path in the state graph) and the size of the Büchi automaton representing the negation of the property [CKOS05]. We recorded the output of the ANALYSIS component and the time required to produce the output result. Note that we did not report the time required by the model checker and the proof extraction separately since the cost of executing the model checker is negligible compared to the cost of computing the topological proof.

Results. Our results are reported in Fig. 4. The x-axis reports the number of states of each random model. The y-axis reports the time required by T0rPEDO-MUP and T0rPEDO-SMT for each run that finished within the timeout. T0rPEDO-MUP was not able to finish within the timeout for models above 20 states. For property ϕ_1 , T0rPEDO-MUP was able to process models with 10 and 20 states. For these models, the differences between the time required by T0rPEDO-MUP and the time required by T0rPEDO-SMT are approximately 3min and 36min, respectively. For property ϕ_2 , T0rPEDO-MUP was able to process only models with 10 states. For this model, the difference between the time required by T0rPEDO-MUP and the time required by T0rPEDO-SMT is approximately 2min. T0rPEDO-SMT was able to finish within the timeout for any model containing up to 40 states. The propositional logic formulae generated by T0rPEDO-SMT had approximately 1000 propositional logic operators. For the cases in which both T0rPEDO-MUP and T0rPEDO-SMT finished within the timeout, they required on average 15m and 1.4m, respectively.

The answer to **RQ3** is that, on the considered models, TOrPEDO-SMT can verify within the timeout models which are double in size w.r.t. those which could be verified by TOrPEDO-MUP within the timeout. For the cases in which both the versions of our tool finished within the timeout, TOrPEDO-MUP and TOrPEDO-SMT required on average 15m and 1.4m, Therefore, TOrPEDO-SMT is significantly more efficient than TOrPEDO-MUP.

6.4. Usefulness—RQ4

To assess how useful TOrPEDO is in supporting the evaluation of design choices, we considered a (small) model example from the genomic domain, related to Gene Regulatory Networks (GRNs). GRNs are collections of molecular regulators, interacting with each other and governing the gene expression levels of mRNA and proteins. Typically, GRNs are represented using nodes (genes) connected by edges (inhibition/activation actions); edges may be weighted or unweighted using some coefficient (e.g., inferred using the Banjo method [YSW⁺04]). GRNs are deduced from gene expression [Alu05] or ChIP-seq experiments data [SPM19] and are used to understand fundamental biological processes in the cell, as well as the pathogenesis of some diseases [ESDHK14, LXZW12], using genomic data integrated from several sources [BCMC21].

Verifying whether GRNs meet certain properties is a widely recognized problem [JCL⁺09]. The correctness of the inferred networks may be performed manually (by comparison with public database, e.g., KEGG [KG00] and GO [ABB⁺00]) or automatically (e.g., by using symbolic model checking techniques [GKD⁺15, MDKG15]). We evaluated how TOrPEDO supports designers in establishing appropriate models of GRNs.

The MAPK Example. We considered a small network of the MAPK pathway⁵, inspired by [GKD⁺15] and translated it into a PKS. Each of the six encompassed genes is represented by one proposition (i.e., *Msg5*, *Fus3*, *Ste7*, *Far1*, *Ste11*, and *Dig1/2*). The proposition is true if the gene is activated, false otherwise. Each configuration of the network describes the status of all the genes (activated/deactivated) and is represented as a state in the PKS. In total, we have 64 (2^6) states that represent all possible statuses of the genes. Transitions among PKS states encode how the status of the genes can change according to the behavior specified by the regulatory network. As initially the network can be in any configuration, all states of the PKS are initial states.

We considered two LTL properties that MAPK pathway should satisfy (i.e., ϕ_1 and ϕ_2) and one (ϕ_3) that it should not satisfy. These properties are inspired by the CTL specifications provided in [GKD⁺15] and abstracted from KEGG's pathways characteristics [KG00], and then discussed with domain experts. Property $\phi_1 = Fus3 \rightarrow \mathcal{X}(\neg Dig1/2)$ is expressing that if *Fus3* is activated, *Dig1/2* will be inhibited immediately in the next step (i.e., *Fus3* is a direct inhibitor of *Dig1/2*). Property $\phi_2 = \mathcal{G}(Msg5 \vee Fus3) \rightarrow \mathcal{F}(\neg Ste11)$ means that if globally either *Msg5* or *Fus3* is activated, finally *Ste11* will be inhibited. Finally, property $\phi_3 = \mathcal{G}(Ste7 \rightarrow \mathcal{F}(Fus3))$ is checking whether or not *Ste7*'s activation will finally inhibit or promote the transcription of *Fus3* cell cycle regulatory gene.

Methodology. As the initial model, we considered a version of the PKS having multiple uncertainty points, specifically: on *Dig1/2* in two different states, on *Ste11* in one state, and on *Msg5* in another state. For the three observed properties, we simulated an incremental model design by running TOrPEDO on the initial model and we assess how useful are the artifacts produced by TOrPEDO to guide the model design. We articulate the timeline of experiments in Table 9.

⁵ The MAPK pathway is a set of chained proteins communicating a signal from a receptor on the cell surface to the DNA stored in the cell nucleus.

Table 9. LTL formulas checked on the 64 states (P)KS representing a sub-network of MAPK pathway.

Property	Initial Model	Revision 1	Revision 2	Revision 3	Revision 4
ϕ_1	ANALYSIS = \top	RE-CHECK = \top	RE-CHECK = \top	ANALYSIS = \top	ANALYSIS = \top
ϕ_2	ANALYSIS = ?	RE-CHECK = ?	RE-CHECK = ?	ANALYSIS = \top	ANALYSIS = \top
ϕ_3	ANALYSIS = \perp	RE-CHECK = \perp	RE-CHECK = \perp	ANALYSIS = \perp	ANALYSIS = \top

- **Step 1.** First, we run the ANALYSIS phase on the model against all three requirements, obtaining results \top for ϕ_1 , ? for ϕ_2 and \perp for ϕ_3 .
- **Step 2.** We then inspected the topological proofs generated for ϕ_1 and ϕ_2 and, after consulting with domain experts, we proposed a first revision of the model that did not conflict with the listed clauses: namely, we assigned $Msg5 = \perp$ in states not shown in the proof. After this revision, it was sufficient to run the RE-CHECK procedure, to confirm the previous verification results.
- **Step 3.** Similarly, by inspecting again the topological proofs, we proposed a second change in the model, producing a second revision that included new transitions among given states. In particular, adding new transitions allows us to envision a wider set of changes of configurations between the states of the GRN. In other words, we allowed additional gene activations and inhibitions with respect to the initially considered set. As this change does not involve any part of the model that is mentioned in the topological proof, we could again confirm the previous results with a simple syntactic RE-CHECK of TOrPEDO.
- **Step 4.** With the purpose of satisfying property ϕ_2 , we produced a Revision 3 which included refinements of the *Ste11* proposition to the value \perp . Such change impacted clauses specified in the topological proof. As such, we had to re-execute the ANALYSIS phase and obtained, in conclusion, that also property ϕ_2 was satisfied by the third revised instance of the model.
- **Step 5.** Finally, we also inspected the counterexample obtained by running TOrPEDO on the third revised model against ϕ_3 and, by changing the truth value of a number of propositions in the model we produced a Revision 4. We run the ANALYSIS phase and obtained a \top result also for the last property.

Results. This exemplifying iterative design process demonstrated that we were able to evaluate three properties on five different models (an initial one, plus four different revisions). We evaluated each of these properties by only running the analysis three times, while using a simple syntactic check twice. The topological proofs provide useful information to identify the parts of the models to be changed. They effectively enabled us to identify the portions of the model that influenced the satisfaction of the properties of interest. This information was useful for designing the model revisions. For example, in Step 2 we changed the value of a proposition. Since this proposition was not involved in any of the clauses of the topological proof, we could safely change its value. The RE-CHECK component was only executed for confirmation. Similarly, in Step 3 we applied another change to the model, this time by adding new transitions among given states. As for the previous step, our change did not involve any part of the model that was mentioned in the topological proof. Again, we verified our change by running the RE-CHECK component. In these two cases, the RE-CHECK phase confirmed that the revisions were compliant with the topological proofs and avoided re-executing the model checker.

Our experiment showed that the information contained in the topological proof was useful in our model design as it effectively guided us during the creation of the model revisions of a (small) model example from the genomic domain, related to Gene Regulatory Networks (GRNs).

The answer to **RQ4** is that the topological proofs and counterexamples provided by TOrPEDO effectively supported the development of a (P)KS representing a gene regulatory network.

7. Related work

Partial knowledge has been considered in requirement analysis and elicitation [MSG17, LKMU08, CSV⁺19], in novel robotic planners [MGPT18], software models [UBC09, UABD⁺13, FSC12, AGC12, MSCG18, MSCG19], and testing [DHKN14, Tre99, vdBRT04, CBGS18, CGS20]. Several researchers analyzed the model checking

problem for partially specified systems [MSG16, CDEG04], considering both three-valued [LT88, GHJ01, BG99, BG00, GP11] and multi-valued [GC03a, BG04] semantics. Other works apply model checking to incremental program development [HJMS03, BHJM07, JHC19]. However, all these model checking approaches do not provide an *explanation* on why a property is satisfied, by means of a *certificate* or *proof*. Although several works have tackled this problem [BMS⁺17, TC02, PZ01, PPZ01, GRT18, DN17], differently from this work, they mostly aim to automate proof reproducibility.

Tao and Li [TL17] propose a theoretical solution to model repair: the problem of finding the minimum set of states in a KS which makes a formula satisfiable. However, the problem is different from the one addressed in this paper. Furthermore, the framework is only theoretical and based on complete systems.

Approaches were proposed in the literature to provide explanations by using different artifacts. For example, other works commonly use a different notion of witnesses: a path of the model that satisfies (or possibly satisfies) a property of interest [TG19, BCC⁺99, HLSU02, Nam01, TGNB20]. In our work, we proposed topological proofs, a new type of witnesses that is significantly different from the one presented in the literature. Other works (e.g., [GC03b, SG03]) studied how to enrich counterexamples with additional information in a way that allows better understanding the property violation. Work has also been done to generate abstractions of the counterexamples that are easier to understand (e.g., [EMA10]). Alur et al. [AMT13] analyzed the problem of synthesizing a controller that satisfies a given specification. When the specification is not realizable, a counter-strategy is returned as a witness. Pencolé et al. [PSMTM17] analyzed model consistency, i.e., the problem of checking whether the system run-time behaviour is consistent with a formal specification. Bernasconi et al. [BMS⁺17] proposed an approach that combines model checking and deductive proofs in a multi-valued context. The notion of topological proof proposed in this work is substantially different from the notion of deductive proof.

The works on vacuity checking (e.g., [FKSFV08, MS20, SDGC10]) are also related to our work. Specifically, a property ϕ is vacuously satisfied on a model M if it has a subformula ϕ' that does not affect the satisfaction of ϕ in M . Other works (e.g., [PQ13, RLF⁺13, SDGC10]) studied how to understand why a property is unsatisfiable. These problems are different from the one considered in this paper, where the goal is to provide a slice of the original model that preserves the (possible) satisfaction of the property.

8. Conclusions

We have proposed TOrPEDO, an integrated framework that supports the iterative creation of model revisions. The framework provides a guide for the designer who wishes to preserve slices of her model that contribute to satisfying fundamental requirements while other parts of the model are modified. For these purposes, the notion of topological proof has been formally and algorithmically described. This corresponds to a set of constraints that, if kept when changing the proposed model, ensure that the behavior of the model w.r.t. the property of interest is preserved. Our Lemmas and Theorems prove the soundness of our framework, i.e., how it preserves correctness in the case of PKS and LTL. The proposed framework can be used as a baseline for other FM frameworks, and can be extended by considering other modeling formalisms that can be mapped onto PKSs.

We presented two implementations of TOrPEDO, namely TOrPEDO-MUP and TOrPEDO-SMT. TOrPEDO-MUP is our initial implementation [MRB20] of TOrPEDO that uses PLTL-MUP to extract topological proofs. With the intent of providing practical and efficient support for flexible model design and wider adoption of our framework, in this work, we proposed TOrPEDO-SMT. TOrPEDO-SMT uses SMT techniques to extract topological proofs.

TOrPEDO was evaluated by showing the effectiveness of the ANALYSIS and RE-CHECK algorithms included in the framework. Results showed that proofs are smaller than the original models, and can be verified in most of the cases using a simple syntactic check. Additionally, we analyzed the efficiency of TOrPEDO and compared TOrPEDO-MUP with TOrPEDO-SMT. Our results show that TOrPEDO produces a topological proof within two hours for models with less than 40 states. Furthermore, our results show that TOrPEDO-SMT is more efficient than TOrPEDO-MUP. Note that, the ANALYSIS phase of TOrPEDO combines three-valued model checking and existing tools for the UCs computation. Therefore, its scalability improves as the performance of these frameworks enhances. Finally, we assessed how useful is TOrPEDO in supporting the evaluation of alternative design choices of (small) model instances in applied domains. Our results show that the topological proofs and counterexamples provided by TOrPEDO effectively supported the development of a model of a gene regulatory network.

Acknowledgements

This work has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant No 694277).

We thank the anonymous reviewers for the useful comments.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

References

- [ABB⁺00] Ashburner M, Ball CA, Blake JA, Botstein D, Butler H, Cherry JM, Davis AP, Dolinski K, Dwight SS, Eppig JT, et al (2000) Gene ontology: tool for the unification of biology. *Nat Genet* 25(1):25–29
- [AGC12] Albarghouthi A, Gurfinkel A, Chechik M (2012) From under-approximations to over-approximations and back. In: International conference on tools and algorithms for the construction and analysis of systems, pp 157–172. Springer
- [Alu05] Aluru S (2005) Handbook of computational molecular biology. Chapman & Hall/CRC
- [AMT13] Alur R, Moarref S, Topcu U (2013) Counter-strategy guided refinement of GR(1) temporal logic specifications. In: Formal methods in computer-aided design, pp 26–33. IEEE
- [BCC⁺99] Biere A, Cimatti A, Clarke EM, Fujita M, Zhu Y (1999) Symbolic model checking using SAT procedures instead of BDDs. In: Design automation conference, pp 317–320. ACM
- [BCE⁺06] Brunet G, Chechik M, Easterbrook S, Nejati S, Niu N, Sabetzadeh M (2006) A manifesto for model merging. In: International workshop on Global integrated model management, pages 5–12. ACM
- [BCMC21] Bernasconi A, Canakoglu A, Masseroli M, Ceri S (2021) The road towards data integration in human genomics: players, steps and interactions. *Brief Bioinform* 22(1):30–44
- [BG99] Bruns G, Godefroid P (1999) Model checking partial state spaces with 3-valued temporal logics. In: International conference on computer aided verification, pp 274–287. Springer
- [BG00] Bruns G, Godefroid P (2000) Generalized model checking: reasoning about partial state spaces. In: International conference on concurrency theory, pp 168–182. Springer
- [BG04] Bruns G, Godefroid P (2004) Model checking with multi-valued logics. In: International colloquium on automata, languages and programming, pp 281–293. Springer
- [BHJM07] Beyer D, Henzinger TA, Jhala R, Majumdar R (2007) The software model checker Blast. *Int J Softw Tools Technol Transf* 9(5–6):505–525
- [BK08] Baier C, Katoen J-P (2008) Principles of model checking. The MIT Press
- [BKR15] Baresi L, Kallehbasti MMP, Rossi M (2015) Efficient scalable verification of LTL specifications. In: International conference on software engineering, pp 711–721. IEEE
- [BMS⁺17] Bernasconi A, Menghi C, Spoletini P, Zuck LD, Ghezzi C (2017) From model checking to a temporal proof for partial models. In: International conference on software engineering and formal methods, pp 54–69. Springer
- [CBGS18] Camilli M, Bellettini C, Gargantini A, Scandurra P (2018) Online model-based testing under uncertainty. In: International symposium on software reliability engineering (ISSRE), pp 36–46. IEEE
- [CCG⁺02] Cimatti A, Clarke E, Giunchiglia E, Giunchiglia F, Pistore M, Roveri M, Sebastiani R, Tacchella A (2002) NuSMV 2: an opensource tool for symbolic model checking. In: International conference on computer aided verification, pp 359–364. Springer
- [CDEG04] Chechik M, Devereux B, Easterbrook S, Gurfinkel A (2004) Multi-valued symbolic model-checking. *Trans Softw Eng Methodol* 12(4):1–38
- [CGS20] Camilli M, Gargantini A, Scandurra P (2020) Model-based hypothesis testing of uncertain software systems. *Softw Test Verif Reliab* 30(2):e1730
- [CKOS04] Clarke E, Kroening D, Ouaknine J, Strichman O (2004) Completeness and complexity of bounded model checking. In: International conference on verification, model checking, and abstract interpretation, pp 85–96. Springer
- [CKOS05] Clarke E, Kroening D, Ouaknine J, Strichman O (2005) Computational challenges in bounded model checking. *Int J Softw Tools Technol Transf* 7(2):174–183
- [CSV⁺19] Chechik M, Salay R, Viger T, Kokaly S, Rahimi M (2019) Software assurance in an uncertain world. In: Fundamental approaches to software engineering, pp 3–21
- [DAC99] Dwyer MB, Avrunin GS, Corbett JC (1999) Patterns in property specifications for finite-state verification. In: International conference on software engineering, pp 411–420. ACM

- [DHKN14] Daca P, Henzinger TA, Krenn W, Nickovic D (2014) Compositional specifications for ioco testing. In: International conference on software testing, verification and validation, pp 373–382. IEEE
- [DMB08] De Moura L, Björner N (2008) Z3: an efficient smt solver. In: International conference on tools and algorithms for the construction and analysis of systems, pp 337–340. Springer
- [DN17] Deng C, Namjoshi KS (2017) Witnessing network transformations. In: International conference on runtime verification, pp 155–171. Springer
- [EC01] Easterbrook S, Chechik M (2001) A framework for multi-valued reasoning over inconsistent viewpoints. In: International conference on software engineering, pp 411–420. IEEE
- [ECD⁺03] Easterbrook SM, Chechik M, Devereux B, Gurfinkel A, Lai Albert YC, Petrovykh V, Taffiovich A, Thompson-Walsh CD (2003) χ Chek: a model checker for multi-valued reasoning. In: International conference on software engineering, pp 804–805. IEEE
- [EMA10] Een N, Mishchenko A, Amla N (2010) A single-instance incremental SAT formulation of proof- and counterexample-based abstraction. In: Conference on formal methods in computer-aided design, pp 181–188. IEEE
- [ESDHK14] Emmert-Streib F, Dehmer M, Haibe-Kains B (2014) Gene regulatory networks and their applications: understanding biological and medical problems in terms of networks. *Front Cell Dev Biol*, 2
- [eta20] ETAPS 2018 Test of time award. <https://etaps.org/about/test-of-time-award/test-of-time-award-2018>, 07 2020
- [FKSFV08] Fisman D, Kupferman O, Sheinvald-Faragy S, Vardi MY (2008) A framework for inherent vacuity. In: International Haifa verification conference, pp 7–22. Springer
- [FSC12] Famelis M, Salay R, Chechik M (2012) Partial models: Towards modeling and reasoning with uncertainty. In: International conference on software engineering, pp 7–22. IEEE
- [FUMK06] Foster H, Uchitel S, Magee J, Kramer J (2006) LTSA-WS: a tool for model-based verification of web service compositions and choreography. In: International conference on Software engineering, pp 771–774. ACM
- [GC03a] Gurfinkel A, Chechik M (2003) Multi-valued model checking via classical model checking. In: International conference on concurrency theory, pp 263–277. Springer
- [GC03b] Gurfinkel A, Chechik M (2003) Proof-like counter-examples. In: International conference on tools and algorithms for the construction and analysis of systems, pp 160–175. Springer
- [GH05] Godefroid P, Huth M (2005) Model checking vs. generalized model checking: semantic minimizations for temporal logics. In: Logic in computer science, pp 158–167. IEEE
- [GHJ01] Godefroid P, Huth M, Jagadeesan R (2001) Abstraction-based model checking using modal transition systems. In: International conference on concurrency theory, pp 426–440. Springer
- [GJ03] Godefroid P, Jagadeesan R (2003) On the expressiveness of 3-valued models. In: International conference on verification, model checking, and abstract interpretation, pp 206–222. Springer
- [GKD⁺15] Gong H, Klinger J, Damazyn K, Li X, Huang S (2015) A novel procedure for statistical inference and verification of gene regulatory subnetwork. *BMC Bioinform*, 16(7):1–10
- [GP09] Godefroid P, Piterman N (2009) LTL generalized model checking revisited. In: Verification, model checking, and abstract interpretation, pp 89–104. Springer
- [GP11] Godefroid P, Piterman N (2011) LTL generalized model checking revisited. *Int J Softw Tools Technol Transf* 13(6):571–584
- [GRT18] Griggio A, Roveri M, Tonetta S (2018) Certifying proofs for LTL model checking. In: Formal methods in computer aided design, pp 1–9. IEEE
- [HJMS03] Henzinger TA, Jhala R, Majumdar R, Sanvido MAA (2003) Extreme model checking. In: Verification: theory and practice, pp 332–358. Springer
- [HLSU02] Hong HS, Lee I, Sokolsky O, Ural H (2002) A temporal logic based theory of test coverage and generation. In: International conference on tools and algorithms for the construction and analysis of systems, pp 327–341. Springer
- [JCL⁺09] Jha SK, Clarke EM, Langmead CJ, Legay A, Platzer A, Zuliani P (2009) A bayesian approach to model checking biological systems. In: Computational methods in systems biology, pp 218–234. Springer
- [JHC19] Jiang J-M, Hong Z, Chen Y (2019) Modeling and analyzing incremental natures of developing software. *Trans Manag Inf Syst* 10(2)
- [KG96] Kupferman O, Grumberg O (1996) Branching-time temporal logic and tree automata. *Inf Comput* 125(1):62–69
- [KG00] Kanehisa M, Goto S (2000) KEGG: kyoto encyclopedia of genes and genomes. *Nucleic Acids Res* 28(1):27–30
- [KOS⁺11] Kroening D, Ouaknine J, Strichman O, Wahl T, Worrell J (2011) Linear completeness thresholds for bounded model checking. In: Computer aided verification, pp 557–572. Springer
- [Kri63] Kripke SA (1963) Semantical considerations on modal logic. *Acta Philos Fennica* 16(1963):83–94
- [LKMU08] Letier E, Kramer J, Magee J, Uchitel S (2008) Deriving event-based transition systems from goal-oriented requirements models. *Autom Softw Eng*, pp 175–206
- [LT88] Larsen KG, Thomsen B (1988) A modal process logic. In: Logic in computer science, pp 203–210. IEEE
- [LXZW12] Liang X-J, Xia Z, Zhang L-W, Wu F-X (2012) Inference of gene regulatory subnetworks from time course gene expression data. In: BMC bioinformatics, volume 13, page S3. Springer
- [MDKG15] Ma Y, Damazyn K, Klinger J, Gong H (2015) Inference and verification of probabilistic graphical models from high-dimensional data. In: International conference on data integration in the life sciences, pp 223–239. Springer
- [MGPT18] Menghi C, Garcia S, Pelliccione P, Tumova J (2018) Multi-robot LTL planning under uncertainty. In: Formal methods, pp 399–417. Springer
- [MRB20] Menghi C, Rizzi AM, Bernasconi A (2020) Integrating topological proofs with model checking to instrument iterative design. In: Fundamental approaches to software engineering, pp 53–74. Springer
- [MS20] Maoz S, Shalom R (2020) Inherent vacuity for GR(1) specifications. In: European software engineering conference and symposium on the foundations of software engineering, pp 99–110. ACM
- [MSCG18] Menghi C, Spoletini P, Chechik M, Ghezzi C (2018) Supporting verification-driven incremental distributed design of components. In: Fundamental approaches to software engineering, pp 169–188. Springer

- [MSCG19] Menghi C, Spoletini P, Chechik M, Ghezzi C (2019) A verification-driven framework for iterative design of controllers. *Formal Aspects Comput* 31(5):459–502
- [MSG16] Menghi C, Spoletini P, Ghezzi C (2016) Dealing with incompleteness in automata-based model checking. In: *Formal Methods*, pp 531–550. Springer
- [MSG17] Menghi C, Spoletini P, Ghezzi C (2017) Integrating goal model analysis with iterative design. In: *International working conference on requirements engineering: foundation for software quality*, pp 112–128. Springer
- [Nam01] Namjoshi KS (2001) Certifying model checkers. In: *Computer aided verification*, pp 2–13. Springer
- [PKRB20] Pourhashem KMM, Rossi MG, Baresi L (2020) On how bit-vector logic can help verify LTL-based specifications. *IEEE Trans Softw Eng*, pp 1–1
- [Pnu77] Pnueli A (1977) The temporal logic of programs. In: *Annual symposium on foundations of computer science*, pp 46–57. IEEE
- [PPZ01] Peled D, Pnueli A, Zuck L (2001) From falsification to verification. In: *Foundations of software technology and theoretical computer science*, pp 292–304
- [PQ13] Pill I, Quaritsch T (2013) Behavioral diagnosis of LTL specifications at operator level. In: *International joint conference on artificial intelligence*, pp 1053–1059. IJCAI/AAAI
- [PSMTM17] Pencolè Y, Steinbauer G, Mühlbacher C, Travè-Massuyès L (2017) Diagnosing discrete event systems using nominal models only. In: *International workshop on principles of diagnosis*, pp 169–183. EasyChair
- [PZ01] Peled D, Zuck L (2001) From model checking to a temporal proof. In: *International SPIN workshop on model checking of software*, pp 1–14. Springer
- [RLF⁺13] Raman V, Lignos C, Finucane C, Lee KCT, Marcus MP, Kress-Gazit H (2013) Sorry Dave, I'm Afraid I Can't Do That: explaining unachievable robot tasks using natural language. In: *Robotics: science and systems*, vol 2, pp 2–1. Citeseer
- [SDGC10] Simmonds J, Davies J, Gurfinkel A, Chechik M (2010) Exploiting resolution proofs to speed up LTL vacuity detection for BMC. *Int J Softw Tools Technol Transf* 12(5):319–335
- [SG03] Shoham S, Grumberg O (2003) A game-based framework for ctl counterexamples and 3-valued abstraction-refinement. In: *International conference on computer aided verification*, pp 275–287. Springer
- [SGT13] Sergeant T, Goré SR, Thomson J (2013) Finding minimal unsatisfiable subsets in linear temporal logic using BDDs, https://cs.anu.edu.au/courses/csprojects/13S1/Reports/Timothy_Sergeant_Report.pdf.
- [SHH12] Sais L, Hacid M-S, Hanry F (2012) On the complexity of computing minimal unsatisfiable LTL formulas. *Electronic Colloquium on Computational Complexity (ECCC)*, 19:69, Hasso Plattner Institute
- [sig20] SIGPLAN ACM - Programming Languages Software Award. <http://www.sigplan.org/Awards/Software/>, 07 2020
- [SLJ⁺06] Schuppan V, Latvala T, Junttila T, Heljanko K, Biere A (2006) Linear encodings of bounded LTL model checking. *Log Methods Comput Sci*, 2, Episciences.org
- [SPM19] Stamoulakatou E, Piccardi C, Masseroli M (2019) Analysis of gene regulatory networks inferred from chip-seq data. In: *International work-conference on bioinformatics and biomedical engineering*, pp 319–331. Springer
- [TC02] Tan L, Cleaveland R (2002) Evidence-based model checking. In: *International conference on computer aided verification*, pp 455–470. Springer
- [TG19] Timm N, Gruner S (2019) Abstraction refinement with path constraints for 3-valued bounded model checking. In: *Formal techniques for safety-critical systems*, pp 139–157. Springer
- [TGNB20] Timm N, Gruner S, Nxumalo M, Botha J (2020) Model checking safety and liveness via k-induction and witness refinement with constraint generation. *Science of Computer Programming*, 200:102532, Elsevier
- [TL17] Tao X, Li G (2017) The complexity of linear-time temporal logic model repair. In: *International workshop on structured object-oriented formal language and method*, pp 69–87. Springer
- [Tor20] Torpedo. <http://github.com/alessandrizzit/torpedo>, 2020
- [Tre99] Tretmans J (1999) Testing concurrent systems: a formal approach. In: *International conference on concurrency theory*, pp 46–65. Springer
- [UABD⁺13] Uchitel S, Alrajeh D, Ben-David S, Braberman V, Chechik M, De Caso G, D'Ippolito N, Fischbein D, Garbervetsky D, Kramer J et al (2013). Supporting incremental behaviour model elaboration. *Comput Sci-Res Dev* 28(4):279–293
- [UBC09] Uchitel S, Brunet G, Chechik M (2009) Synthesis of partial behavior models from properties and scenarios. *IEEE Trans Softw Eng*, 35(3):384–406
- [Uch09] Uchitel S (2009) Partial behaviour modelling: foundations for incremental and iterative model-based software engineering. In: *Formal methods: foundations and applications*. Springer
- [vdBRT04] van der Bijl M, Rensink A, Tretmans J (2004) Compositional testing with ioco. In: *Formal approaches to software testing*, pp 86–100. Springer
- [YSW⁺04] Yu J, Smith VA, Wang PP, Hartemink AJ, Jarvis ED (2004) Advances to bayesian network inference for generating causal networks from observational biological data. *Bioinformatics*, oxford university press, 20(18):3594–3603

Received 14 January 2021

Accepted in revised form 15 July 2021 by Jordi Cabot, Heike Wehrheim and Eerke Boiten

Published online 6 November 2021