Compositional Model Checking for Multi-properties

Ohad Goudsmid[1(✉)], Orna Grumberg[1], and Sarai Sheinvald[2]

[1] Department of Computer Science, The Technion, Haifa, Israel
goudsmidohad@cs.technion.ac.il
[2] Department of Software Engineering, ORT Braude College of Engineering, Karmiel, Israel

Abstract. *Hyperproperties* lift conventional trace properties in a way that describes how a system behaves in its entirety, and not just based on its individual traces. We generalize this notion to *multi-properties*, which describe the behavior of not just a single system, but of a set of systems, which we call a *multi-model*. We demonstrate the usefulness of our setting with practical examples. We show that model-checking multi-properties is equivalent to model-checking hyperproperties. However, our framework has the immediate advantage of being *compositional*. We introduce sound and complete compositional proof rules for model-checking multi-properties, based on over- and under-approximations of the systems in the multi-model. We then describe methods of computing such approximations. The first is abstraction-refinement based, in which a coarse initial abstraction is continuously refined using counterexamples, until a suitable approximation is found. The second, tailored for models with finite traces, finds suitable approximations via the L^* learning algorithm. Our methods can produce much smaller models than the original ones, and can therefore be used for accelerating model-checking for both multi-properties and hyperproperties.

1 Introduction

Temporal logics, such as LTL, are widely used for specifying program behaviors. An LTL property characterizes a set of traces, each of which satisfies the property. It has recently been shown that trace properties are insufficient for characterizing and verifying security vulnerabilities or their absence.

The notion of *hyperproperties* [9], a generalization of trace properties, provides a uniform formalism for specifying properties of *sets of traces*. Hyperproperties are particularly suitable for specifying security properties. For instance, secure information flow may be characterized by identifying low-security variables that may be observable to the environment, and high-security variables that should not be observable outside. Secure information flow is maintained in

This research was partially supported by the Mel Berlin fellowship and partially by the Israel Science Foundation.

© Springer Nature Switzerland AG 2021
F. Henglein et al. (Eds.): VMCAI 2021, LNCS 12597, pp. 55–80, 2021.
https://doi.org/10.1007/978-3-030-67067-2_4

a system if for every two traces, if their low-security inputs are identical then so are their low-security outputs, regardless of the values of high-security variables. This property cannot be characterized via single traces.

While hyperproperties are highly useful, they are still limited: they can only refer to the system as a whole. Systems often comprise several components, and it is desired to relate traces from one component to traces of another. A prominent such example is *diversity* [16]. Diversity generalizes the notion of security policies by considering policies of a set of systems. The systems are all required to implement the same functionality but to differ in their implementation details. As noticed in [9], such a set of policies could, in principle, be modeled as a hyperproperty on a single system, which is a product of all the systems in the set. This, however, is both unnatural and highly inefficient.

We remedy this situation by presenting a framework which explicitly describes the system as a set of systems called a *multi-model*, and provides a specification language, MultiLTL, which explicitly relates traces from the different components in the multi-model. Our framework enables to directly and naturally describe properties like diversity, while avoiding the need for a complex translation.

Our framework also has the immediate advantage of being *compositional*. We thus suggest a sound and complete compositional model-checking rule. The rule is based on abstracting each of the components by over- and under-approximations, thus achieving additional gain.

We then suggest methods of computing such approximations. The first is based on *abstraction-refinement*, in which a coarse initial abstraction is continuously refined by using counterexamples, until a suitable approximation is found. The second, tailored for models with finite traces, finds suitable approximations via the L^* learning algorithm. Our methods can produce much smaller models than the original ones, and can therefore be used for accelerating model-checking for both multi-properties and hyperproperties.

We now describe our work in more detail. Our framework consists of multi-models, which are tuples of Kripke structures. The logic we focus on, called MultiLTL, is an extension of HyperLTL [8]. MultiLTL allows indexed quantifications, $\forall^i$ and $\exists^i$, referring to the i'th component-model in the multi-model.

We show that there is a two-way reduction between the model-checking problem for HyperLTL and the model-checking problem for MultiLTL. We emphasize, that even though the two model-checking problems are equivalent, our new framework is clearly more powerful as it enables a direct specification and verification of the whole system by explicitly referring to its parts.

We exploit this power by introducing two compositional proof rules, which are based on over- and under-approximations for each system component separately. These proof rules are capable of proving a MultiLTL property or its negation for a given multi-model.

We suggest two approaches to computing these approximations for the compositional proof rules. The first approach is based on *abstraction-refinement*. The approximations are computed gradually, starting from coarse approximations

and refined based on counterexamples. The abstraction-refinement approach is implemented using one of two algorithms. In both algorithms, when model-checking the abstract multi-model is successful, we conclude that model-checking for the original multi-model holds. Otherwise, a counterexample is returned.

The first algorithm is based on counterexamples from the multi-model only. For each component-model, we find a behavior that should be eliminated from an over-approximated component-model or added to an under-approximated component-model, and refine the components accordingly.

The second algorithm is applicable for a restricted type of MultiLTL properties, in which the quantification consists of a sequence of $\forall$ quantifiers followed by a sequence of $\exists$ quantifiers. In hyperproperties, this is a useful fragment which allows specifying noninterference and generalized noninterference, observational determinism, and more. The counterexamples in this case come directly from the unsuccessful model-checking process, and therefore refer both to the model and to the property. Notice that, since the abstract component-models are typically much smaller than the original component-models, their model-checking is much faster.

The logics of MultiLTL and the model of Kripke structure are designed for describing and modeling the behavior of on-going systems. However, to do the same for terminating programs with finite traces, a more suitable description is needed. Therefore, we turn our attention to multi-models and multi-properties with finite traces. In this context, we use nondeterministic finite automata (NFA) to describe a system, and a set of NFAs (*multi-NFA*) to describe a set of such systems. For the specification language, we use nondeterministic finite-word hyper-automata (NFH) suggested in [7]. NFH can be thought of as the regular-language counterpart of HyperLTL, and are able to describe the regular properties of sets of finite-word languages, just as HyperLTL is able to describe the properties of a language of infinite traces. Also like HyperLTL, NFH can be easily adjusted to describe multi-properties, a model that we call *multi-NFH*.

We show that, as in the infinite-trace case, there is a two-way reduction between the model-checking problem for NFH and the model-checking problem for multi-NFH. We then proceed to present a compositional model-checking framework for multi-NFH. As in the case of infinite-traces, this framework is based on finding approximations for the NFAs in the multi-model. The method for finding these approximations for this case, however, is learning-based.

Learning-based model-checking [15] seeks candidate approximations by running an automata learning algorithm such as L^* [2]. In the L^* algorithm, a *learner* constructs a finite-word automaton for an unknown regular language $\mathcal{L}$, through a sequence of *membership queries* ("is the word w in $\mathcal{L}$?") and *equivalence queries* ("is $\mathcal{A}$ an automaton for $\mathcal{L}$?"), to which it receives answers from a *teacher* who knows the language. The learner continually constructs and submits candidate automata, until the teacher confirms an equivalence query.

In our algorithm, the learner constructs a set of candidate automata in every iteration, one for every NFA in the multi-model. The key idea is treating these candidate automata as candidate approximations. When an equivalence query

is submitted, we (as the teacher) check whether the NFAs that the learner submitted are suitable approximations. If they are not, we return counterexamples to the learner, based on the given multi-NFA, which it uses to construct the next set of candidates. If they are suitable approximations, we model-check the multi-NFA of the approximations against the multi-NFH. Since the automata that the learner constructs are relatively small, model-checking the candidates multi-model is much faster than model-checking the original multi-model.

In [15], the learning procedure aims at learning the *weakest assumption* W, which is a regular language that contains all the traces that under certain conditions satisfy the specification. The construction of W relies on counterexample words provided by the model checking. We can derive such counterexamples for a certain fragment of multi-NFH. Moreover, we define a suitable weakest assumption for this case, prove that it is regular, and use it as a learning goal in an improved algorithm. Both of these improvements – extracting counterexamples from the model-checker, and learning the weakest assumption rather than the model itself – allow for an even quicker convergence of the model-checking process for this type of multi-properties.

Related Work. *Hyperproperties*, introduced in [9], provide a uniform formalism for specifying properties of sets of traces. Hyperproperties are particularly suitable for specifying security properties, such as secure information flow and non-interference. Two logics for hyperproperties are introduced in [8]: HyperLTL and hyperCTL*, which generalize LTL and CTL*, respectively. Other logics for hyperproperties have been studied in [1,5,6,10,13,14,20].

One of the first sound and complete methods for model-checking hyperproperties is called *self-composition* [4]. Self-composition combines several disjoint copies of the same program, allowing to express relationships among multiple traces. This reduces the k-trace hyperproperty model-checking to trace property model-checking. Unfortunately, the size of the product model increases exponentially with the number of copies. Thus, reasoning directly on the product program is prohibitive.

Many approaches have been suggested for dealing with the high complexity of self-composition. Methods to increase the efficiency of SMT solvers for hyperpropery model-checking have been suggested in [3,19], while a generalization of Hoare triplets for safety-hyperproperties has been presented in [18].

Different approaches to avoid the construction of the full product are presented in [17,21]. The former exploits taint analysis or Bounded Model Checking. The latter infers a self-composition function together with an inductive invariant, suitable for verification.

An automata based algorithm for HyperLTL and HyperCTL* is proposed in [12]. It combines self-composition with ideas from LTL model-checking using alternating automata. A representation of hyperproperties in a form of finite-word automata is developed in [11]. This work introduces a canonical automata representation for regular-k-safety hyperproperties, which are only-universally-quantified safety-hyperproperties.

The first representation of general hyperproperties using finite automata is introduced in [7]. This representation, called *hyperautomata*, allows running multiple quantified words on an automaton. The authors show that hyperautomata can express regular hyperproperties and explore the decidability of nonemptiness (satisfiability) and membership (model-checking) problems. Additionally, they describe an L^*-based learning algorithm for some fragments of hyperautomata.

2 Preliminaries

Kripke Structures are a standard model for ongoing finite-state systems.

Definition 1. *Given a finite set of atomic propositions* AP*, a* Kripke structure *is a 4-tuple* $\mathcal{M} = (S, I, R, L)$*, where* S *is a finite set of* states*,* $I \subseteq S$ *is a non-empty set of* initial states*,* $R \subseteq S \times S$ *is a total* transition relation *and* $L : S \to 2^{AP}$ *is a* labeling function*.*

A *path* in $\mathcal{M}$ is an infinite sequence of states $p = s_0, s_1, s_2, \ldots$ such that $(s_i, s_{i+1}) \in R$ for every $i \in \mathbb{N}$. A *trace over* AP is an infinite sequence $\tau \in \left(2^{AP}\right)^{\omega}$. We sometimes refer to a trace as a *word* over 2^{AP}. A *trace property over* AP is a set of traces over AP.

The *trace that corresponds to a path* p is the trace $\tau(p) = \tau_0, \tau_1, \tau_2, \ldots$ in which $\tau_i = L(s_i)$ for every $i \in \mathbb{N}$. Notice that since R is total, there exists an infinite path from every state. We denote by τ^i the trace $\tau_i, \tau_{i+1}, \ldots$.

Given a word $w = w_0, w_1, \cdots \in (2^{AP})^{\omega}$, a *run of* $\mathcal{M}$ *on* w is a path $p = s_0, s_1 \ldots$ in $\mathcal{M}$ such that $L(s_n) = w_n$ for every $n \in \mathbb{N}$. The *language* $\mathcal{L}(\mathcal{M})$ of $\mathcal{M}$ is the set of all traces corresponding to paths in $\mathcal{M}$ that start in I. The *prefix language* $\mathcal{L}_f(\mathcal{M})$ of $\mathcal{M}$ is the set of all finite prefixes of traces in $\mathcal{L}(\mathcal{M})$. For two Kripke structures $\mathcal{M}, \mathcal{M}'$, we write $\mathcal{M} \models \mathcal{M}'$ to denote that $\mathcal{L}(\mathcal{M}) \subseteq \mathcal{L}(\mathcal{M}')$.

The following is a known result, which can be proven by König's Lemma.

Lemma 1. *For Kripke structures* $\mathcal{M}$ *and* $\mathcal{M}'$*, it holds that* $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{M}')$ *iff* $\mathcal{L}_f(\mathcal{M}) = \mathcal{L}_f(\mathcal{M}')$*.*

2.1 Hyperproperties and HyperLTL

Trace properties and the logics that express them are commonly used to describe desirable system behaviors. However, some behaviors cannot be expressed by referring to each trace individually. In [9], properties describing the behavior of a combination of traces are formalized as *hyperproperties*. Thus, a hyperproperty is a set of sets of traces: all sets that behave according to the hyperproperty.

HyperLTL [8] is an extension of *linear temporal logic* (LTL), a widely used temporal logic for trace properties, to hyperproperties. The formulas of HyperLTL are given by the following grammar:

$$\varphi ::= \exists \pi.\ \varphi \mid \forall \pi.\ \varphi \mid \psi$$

$$\psi ::= a_\pi \mid \neg\psi \mid \psi \vee \psi \mid \mathsf{X}\psi \mid \psi \mathsf{U} \psi \qquad \text{for every } a \in AP$$

Intuitively, $\exists\pi.\varphi$ means that there exists a trace that satisfies φ and $\forall\pi.\varphi$ means that φ holds for every trace. a_π means that a holds in the first state of π. The semantics of X, U and the Boolean operators are similar to those in LTL: $\mathsf{X}\psi$ means that ψ holds in the next state and $\psi_1 \mathsf{U} \psi_2$ means that ψ_1 holds until ψ_2 holds. Based on these operators we define additional operators commonly defined in LTL: $\mathsf{F}\psi$ means that ψ holds eventually and $\mathsf{G}\psi$ means that ψ holds throughout the entire trace.

The semantics of HyperLTL is defined as follows. Let $T \subseteq (2^{AP})^\omega$ be a set of traces over AP, let $\mathcal{V}$ be a set of *trace variables*, and $\Pi : \mathcal{V} \to T$ be a trace assignment. Let $\Pi[\pi \to t]$ be the function obtained from Π, by mapping π to t. Let Π^i be the function defined by $\Pi^i(\pi) = (\Pi(\pi))^i$.

$$
\begin{aligned}
&\Pi \models_T \exists\pi.\psi \quad \text{iff there exists } t \in T \text{ such that } \Pi[\pi \to t] \models_T \psi \\
&\Pi \models_T \forall\pi.\psi \quad \text{iff for every } t \in T,\ \Pi[\pi \to t] \models_T \psi \\
&\Pi \models_T a_\pi \quad \text{iff } a \in \Pi(\pi)[0] \\
&\Pi \models_T \neg\varphi \quad \text{iff } \Pi \not\models_T \varphi \\
&\Pi \models_T \varphi_1 \vee \varphi_2 \quad \text{iff } \Pi \models_T \varphi_1 \text{ or } \Pi \models_T \varphi_2 \\
&\Pi \models_T \mathsf{X}\varphi \quad \text{iff } \Pi^1 \models_T \varphi \\
&\Pi \models_T \varphi_1 \mathsf{U} \varphi_2 \quad \text{iff there exists } i \geq 0 \text{ such that } \Pi^i \models_T \varphi_2 \\
&\qquad\qquad\qquad\qquad \text{and for all } 0 \leq j < i,\ \Pi^j \models_T \varphi_1
\end{aligned}
$$

Notice that when all trace variables of a HyperLTL formula $\mathbb{P}$ are in the scope of a quantifier (i.e, when $\mathbb{P}$ is *closed*), then the satisfaction is independent of the trace assignment, in which case we write $T \models \mathbb{P}$. Given a Kripke structure $\mathcal{M}$ and a HyperLTL formula $\mathbb{P}$, the *model-checking problem* is to decide whether $\mathcal{L}(\mathcal{M}) \models \mathbb{P}$ (which we denote by $\mathcal{M} \models \mathbb{P}$).

By abuse of notation, given traces $w_1, \ldots, w_k$ over AP, we write $\langle w_1, \ldots, w_k \rangle \models \mathbb{Q}_1\pi_1 \ldots \mathbb{Q}_k\pi_k\psi(\pi_1, \ldots, \pi_k)$ if $\Pi \models \psi(\pi_1, \ldots, \pi_k)$, where $\Pi(\pi_i) = w_i$.

3 Multi-models and Multi-properties

We generalize hyperproperties to *multi-properties*, which reason about the connections between several models, which we call a *multi-model.*

Definition 2. *Given* $k \in \mathbb{N}$*, a* k-multi-model *is a* k*-tuple* $\mathbb{M} = \langle \mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_k \rangle$ *of Kripke structures over a common set of atomic propositions* AP*. A* k-multi-property *is a set of tuples* $\mathbb{P} \subseteq (2^{(2^{AP})^\omega})^k$*.*

$\mathbb{M}$ *is a* multi-model *if it is a* k*-multi-model for some* k*, and similarly* $\mathbb{P}$ *is a* multi-property*.*

Intuitively, in a multi-property $\mathbb{P}$, every $T \in \mathbb{P}$ is a tuple of k sets of traces, each interpreted in a model.

We now present MultiLTL, a logic for describing multi-properties. A MultiLTL formula is interpreted over a multi-model $\mathbb{M} = \langle \mathcal{M}_1, \ldots, \mathcal{M}_k \rangle$. We use $[a, b]$, where $a \leq b$ are integers, to denote the set $\{a, a+1, \ldots, b\}$. MultiLTL formulas are defined inductively as follows.

$$\varphi ::= \exists^j \pi.\ \varphi \mid \forall^j \pi.\ \varphi \mid \psi \qquad \text{where } j \in [1, k]$$
$$\psi ::= a_\pi \mid \neg\psi \mid \psi \vee \psi \mid \mathsf{X}\psi \mid \psi \mathsf{U} \psi$$

The only difference in syntax from HyperLTL is that trace quantifiers are now indexed. This index is taken from the set $[1, k]$ for some $k \in \mathbb{N}$. The formula $\exists^j \pi.\varphi$ means that there exists a trace in $\mathcal{M}_j$ that satisfies φ and $\forall^j \pi.\varphi$ means that φ holds for every trace in $\mathcal{M}_j$.

The semantics of MultiLTL is defined as follows. Let $\mathbb{T} = \langle T_1, \ldots T_k \rangle$ be a multi-model over AP. Let $\mathcal{V}$ be a set of trace quantifiers, and let $\Pi : \mathcal{V} \to \bigcup_{i \in [1,k]} T_i$.

$$\Pi \models_{\mathbb{T}} \exists^i \pi.\psi \quad \text{iff there exists } t \in T_i \text{ such that } \Pi[\pi \to t] \models_{\mathbb{T}} \psi$$
$$\Pi \models_{\mathbb{T}} \forall^i \pi.\psi \quad \text{iff } \Pi[\pi \to t] \models_{\mathbb{T}} \psi \text{ for every } t \in T_i$$

The semantics of the temporal operators is defined as in HyperLTL. Since every MultiLTL formula describes a multi-property, we refer to the formulas themselves as multi-properties.

As with HyperLTL, when a MultiLTL formula $\mathbb{P}$ is closed, satisfaction is independent of Π, and we denote $\mathbb{M} \models \mathbb{P}$ for a multi-model $\mathbb{M}$. The *model-checking problem* for MultiLTL is to decide whether $\mathbb{M} \models \mathbb{P}$.

For a MultiLTL formula $\mathbb{P} = \mathbb{Q}^1_{i_1} \ldots \mathbb{Q}^n_{i_n} \varphi$, we define $I_\exists(\mathbb{P}) = \{i \mid \mathbb{Q}^i_{i_j} = \exists \text{ and } i_j \in [1, n]\}$, and $I_\forall(\mathbb{P}) = \{i \mid \mathbb{Q}^i_{i_j} = \forall \text{ and } i_j \in [1, n]\}$. We write $I_\exists$ and $I_\forall$ when $\mathbb{P}$ is clear from the context.

3.1 Examples

We demonstrate the usefulness of MultiLTL and multi-models with several examples. The multi-models we consider consist of models that interact with each other via an asynchronous communication channel (which is not modeled). This assumption is not necessary outside the scope of the examples, where other forms of interactions across models can take place (e.g., shared variables).

Example 1. Consider a multi-model consisting of a client model C and a server model S. We would like to check whether $\langle C, S \rangle \models \forall^C \pi_1 \forall^S \pi_2.\mathsf{G}(r_sent_{\pi_1} \to \mathsf{F} r_received_{\pi_2})$. In this formula, $r_sent_{\pi_1}$ means that a request is sent in C and $r_received_{\pi_2}$ means that a request is received in S. The formula specifies that for every run of the client and for every run of the server, every request sent by the client is eventually received by the server. This is a form of a liveness property that specifies that messages are guaranteed to eventually arrive at their destination. Note that, whether this property holds or not depends in fact on the reliability of the asynchronous communicating channel, connecting the client and the server.

Example 2. Consider again the multi-model of Example 1. Assume that the interaction between the client and the server is as follows. At the beginning of the interaction, the client sends its username and password to the server. Immediately afterwards the server updates its authentication flag and informs the client whether the authentication was successful or not. The client gets this notification one clock cycle after the server authentication flag has been updated. Consider the specification $\mathbb{P}_2$.

$$\mathbb{P}_2 = \forall^S \pi_1 \exists^C \pi_2 \forall^C \pi_3.\ (userDB_{\pi_1} = user_{\pi_2}) \wedge (passDB_{\pi_1} = pass_{\pi_2}) \wedge (\mathsf{X} aut_{\pi_1} \wedge \mathsf{XX} aut_{\pi_2}) \\ \wedge\ ((userDB_{\pi_1} = user_{\pi_3}) \wedge ((passDB_{\pi_1} \neq pass_{\pi_3})) \rightarrow (\mathsf{X}\neg aut_{\pi_1} \wedge \mathsf{XX}\neg aut_{\pi_3})$$

The first line of $\mathbb{P}_2$ states that for every trace of the server there is a trace of the client whose username and password match the username and password in the server database. If so, the authentication succeeds. The second line assures that for each username in the server database there is only one valid password with which the authentication succeeds.

Note that in this example, we describe a property which cannot be described using LTL. Further, it cannot be expressed naturally in HyperLTL. MultiLTL, which explicitly refers to traces in different models within a multi-model, naturally expresses it.

Example 3. We demonstrate again the power of MultiLTL to *naturally* express properties that are not naturally expressible in HyperLTL. *Diversity* [16] refers to security policies of a set of systems. The systems constitute different implementations of the same high-level program. They differ in their implementation details[1], but are equivalent with respect to the input-output they produce. In [16], diversity has been advocated as a successful way to resist attacks that exploit memory layout or instruction sequence specifics.

Assume that we are given a high-level program P and two low-level implementations M_1 and M_2. The following MultiLTL properties describe the fact that all implementations are equivalent to P.

$$\mathbb{P}_1 = \forall^P \pi \exists^{M_1} \pi_1 \exists^{M_2} \pi_2.(input_\pi = input_{\pi_1} = input_{\pi_2}) \wedge \\ \mathsf{G}(end_\pi \wedge end_{\pi_1} \wedge end_{\pi_2} \rightarrow output_\pi = output_{\pi_1} = output_{\pi_2})$$

$$\mathbb{P}_2 = \forall^{M_1} \pi_1 \exists^P \pi.(input_{\pi_1} = input_\pi) \wedge \mathsf{G}(end_{\pi_1} \wedge end_\pi \rightarrow output_{\pi_1} = output_\pi)$$

$$\mathbb{P}_3 = \forall^{M_2} \pi_2 \exists^P \pi.(input_{\pi_2} = input_\pi) \wedge \mathsf{G}(end_{\pi_2} \wedge end_\pi \rightarrow output_{\pi_2} = output_\pi)$$

Note that these properties cannot naturally be expressed in HyperLTL since they require an explicit reference to the models from which the related traces are taken.

[1] For instance, the call stack of procedures is obfuscated by changing the order of variables, the specific memory location of arguments and local variables, etc. The obfuscations differ in the different implementations.

3.2 Model-Checking MultiLTL

We now show that although MultiLTL is a generalization of HyperLTL, the model-checking problems for these logic types is equivalent.

For the first direction, it is easy to see that the model-checking problem for a model $\mathcal{M}$ and a HyperLTL formula $\mathcal{P}$ is equivalent to the model checking problem for $\langle\mathcal{M}\rangle$ and the MultiLTL formula obtained from $\mathcal{P}$ by indexing all of its quantifiers with the same index 1.

For the other direction, we first introduce some definitions. We use the notation $\uplus$ for disjoint union.

Definition 3. *Given a multi-model* $\mathbb{M} = \langle\mathcal{M}_1, \ldots, \mathcal{M}_k\rangle$ *over* AP*, its* union model *denoted* $\cup\mathbb{M}$ *is* $(\uplus_{i=1}^n S_i, \uplus_{i=1}^n I_i, \uplus_{i=1}^n R_i, L)$*, where* $L(s) = L_i(s) \uplus \{\mathbf{i}\}$ *for every* i *and* $s \in S_i$*.*

The indexing by i *of a trace* $\tau = t_0, t_1, \ldots$ *over* AP *is the trace* $\mathsf{ind}_i(\tau) = t_0 \cup \{\mathbf{i}\}, t_1 \cup \{\mathbf{i}\}, \ldots$

Notice that for a trace τ and a multi-model $\mathbb{M} = \langle\mathcal{M}_1, \ldots, \mathcal{M}_k\rangle$, it holds that $\tau \in \mathcal{L}(\mathcal{M}_i) \iff \mathsf{ind}_i(\tau) \in \mathcal{L}(\cup\mathbb{M})$.

Theorem 1. *The model-checking problem for* MultiLTL *is polynomialy reducable to the model-checking problem for* HyperLTL*.*

Proof Sketch. Let $\mathbb{M} = \langle\mathcal{M}_1, \ldots, \mathcal{M}_n\rangle$ be a multi-model over AP, and $\mathbb{P} \in$ MultiLTL. We assume that $\mathbb{P}$ is of the form $\mathbb{Q}_1^1\pi_1 \ldots \mathbb{Q}_n^n\pi_n\varphi$, where φ is in negation normal form. Note that this means that each model is quantified exactly once[2]. Define $\mathcal{M} = \cup\mathbb{M}$. Each (indexed) trace in $\mathcal{M}$ corresponds to one model in $\mathbb{M}$ by its index. Let $\mathcal{P} = \mathbb{Q}_1\pi_1 \ldots \mathbb{Q}_n\pi_n\varphi'$, where φ' is obtained from φ by applying the following changes: for every $a \in AP$, we replace every occurrence of a literal $l = a_\pi$ or $l = \neg a_\pi$ by $\mathbf{i} \rightarrow l$ if π is quantified by $\forall^i$, and by $\mathbf{i} \wedge l$ if π is quantified by $\exists^i$. Intuitively, for $\forall^i$, for every trace $\tau \in \mathcal{M}$, if τ originates from $\mathcal{M}_i$ then we require that τ fulfill the formula and otherwise we require nothing. For $\exists^i$, we require the existence of a trace in $\mathcal{M}$ that originates from $\mathcal{M}_i$ that fulfills the formula. It can be shown by induction that $\mathbb{M} \models \mathbb{P}$ iff $\mathcal{M} \models \mathcal{P}$. □

In [12], the authors presented an algorithm for model-checking HyperLTL that can be easily adjusted for MultiLTL. Thus, there is no need to use the reduction in Theorem 1. The algorithm relies roughly on the repeated intersection of the models under $\exists$ with an automaton for φ, the quantifier-free part of the formula, or, in the case of $\forall$ quantifiers, for $\neg\varphi$ (which involves complementation). Accordingly, the complexity is a tower in the number of models, and the size of the models greatly influences the run-time. In case of a model under $\forall$, a word that is accepted by the intersection is a counterexample for the satisfaction of the $\forall$ requirement. Therefore, in case that the formula $\mathbb{P}$ begins with a sequence of $\forall$ quantifiers followed by a sequence of $\exists$ quantifiers (a fragment which we denote by $\forall^*\exists^*$MultiLTL), it is possible to extract a counterexample for every model under $\forall$ in the multi-model. To summarize, we have the following.

[2] This can be achieved by duplicating components of the multi-model and reordering them so that they match the order of quantification.

Lemma 2. *1. There is a direct algorithm for model-checking* $\mathbb{M} \models \mathbb{P}$.
2. For $\mathbb{P} \in \forall^*\exists^*\mathsf{MultiLTL}$ *with* n *quantifiers such that* $|I_\forall(\mathbb{P})| = k$*, if* $\mathbb{M} \not\models \mathbb{P}$ *then the model-checking algorithm can also extract a counterexample* $\langle w_1, \ldots, w_k \rangle$ *such that* $w_i \in \mathcal{L}(\mathcal{M}_i)$ *for* $i \in [1, k]$*. For* $\langle w_1, \ldots, w_k \rangle$ *it holds that there are no* $w_i \in \mathcal{L}(\mathcal{M}_i)$ *for* $i \in [k+1, n]$ *such that* $\langle w_1, \ldots, w_n \rangle \models \mathbb{P}$.

Note 1. For $\exists$ quantifiers, there is no natural counterexample in the form of a single word. Indeed, a counterexample in this case would need to convince of the lack of existence of an appropriate word.

4 Compositional Proof Rules for Model-Checking MultiLTL

We present two complementing compositional proof rules for the MultiLTL model-checking problem. Let $\mathbb{M}$ be a k-multi-model, and let $\mathbb{P} = \mathbb{Q}_1^{i_1}\pi_1 \ldots \mathbb{Q}_m^{i_m}\pi_m\varphi$ be a MultiLTL formula. The rule (PR) aims at proving $\mathbb{M} \models \mathbb{P}$, and ($\overline{\text{PR}}$) aims at proving the contrary, that is, $\mathbb{M} \models \neg\mathbb{P}$. Every model $\mathcal{A}_i$ in the rules is an *abstraction*. Since some models may be multiply quantified, a model $\mathcal{M}_i$ may have several different abstractions, according to the quantifiers under which $\mathcal{M}_i$ appears in $\mathbb{P}$.

$$\frac{\forall i \in I_\forall.\ \mathcal{M}_{i_j} \models \mathcal{A}_i \quad \forall i \in I_\exists.\ \mathcal{A}_i \models \mathcal{M}_{i_j} \quad \langle \mathcal{A}_1, \ldots, \mathcal{A}_m \rangle \models \mathbb{Q}_1^{i_1}\pi_1 \ldots \mathbb{Q}_m^{i_m}\pi_m\varphi}{\langle \mathcal{M}_1, \ldots, \mathcal{M}_k \rangle \models \mathbb{Q}_1^{i_1}\pi_1 \ldots \mathbb{Q}_m^{i_m}\pi_m\varphi} \tag{PR}$$

$$\frac{\forall i \in I_\forall.\ \mathcal{A}_i \models \mathcal{M}_{i_j} \quad \forall i \in I_\exists.\ \mathcal{M}_{i_j} \models \mathcal{A}_i \quad \langle \mathcal{A}_1, \ldots, \mathcal{A}_m \rangle \models \neg(\mathbb{Q}_1^{i_1}\pi_1 \ldots \mathbb{Q}_m^{i_m}\pi_m\varphi)}{\langle \mathcal{M}_1, \ldots, \mathcal{M}_k \rangle \models \neg(\mathbb{Q}_1^{i_1}\pi_1 \ldots \mathbb{Q}_m^{i_m}\pi_m\varphi)} \tag{$\overline{\text{PR}}$}$$

Intuitively, in (PR), we use an over-approximation for every model under $\forall$, and an under-approximation for every model under $\exists$. The rule ($\overline{\text{PR}}$) behaves dually to (PR) for the negation of $\mathbb{P}$.

Lemma 3. *The proof rules (PR) and ($\overline{\text{PR}}$) are sound and complete.*

Proof Sketch. For completeness, we can choose $\mathcal{A}_i = \mathcal{M}_{i_j}$ for every $i \in [1, m]$. For soundness of (PR), let $\mathcal{A}_1, \ldots, \mathcal{A}_m$ be models for which the premise of (PR) holds. For every universally quantified model $\mathcal{M}_{i_j}$, its abstraction $\mathcal{A}_i$ includes all of its traces (and maybe more). For every existentially quantified model $\mathcal{M}_{i_j}$, a subset of its traces are included in $\mathcal{A}_i$. Therefore, by the semantics of the quantifiers, it is "harder" for each $\mathcal{A}_i$ to satisfy $\mathbb{P}$ than it is for $\mathcal{M}_{i_j}$. Since $\langle \mathcal{A}_1, \ldots, \mathcal{A}_m \rangle \models \mathbb{P}$, we conclude that $\mathbb{M} \models \mathbb{P}$.

For ($\overline{\text{PR}}$), notice that $\neg\mathbb{P} \equiv \overline{\mathbb{Q}}_1^{i_1}\pi_1 \ldots \overline{\mathbb{Q}}_1^{i_m}\pi_m\neg\varphi$, where $\overline{\forall} = \exists$ and $\overline{\exists} = \forall$, conforming to (PR). $\square$

5 Abstraction-Refinement Based Implementation of (PR) and ($\overline{\text{PR}}$)

In this section, we present methods for constructing over- and under-approximations using an abstraction-refinement based approach. We first define the notion of *simulation*.

Definition 4. *Let $\mathcal{M}_1 = (S_1, I_1, R_1, L_1)$ and $\mathcal{M}_2 = (S_2, I_2, R_2, L_2)$ be Kripke structures over AP. A* simulation from $\mathcal{M}_1$ to $\mathcal{M}_2$ *is a relation $H \subseteq S_1 \times S_2$ such that for every $(s_1, s_2) \in H$:*

- $L(s_1) = L(s_2)$
- *For every $(s_1, s_1') \in R_1$ there exists $s_2' \in S_2$ such that $(s_2, s_2') \in R_2$ and $(s_1', s_2') \in H$.*

If additionally, for every $s_0 \in I_1$ there exists $s_0' \in I_2$ such that $(s_0, s_0') \in H$, we denote $\mathcal{M}_1 \leq_H \mathcal{M}_2$. We denote $\mathcal{M}_1 \leq \mathcal{M}_2$ if $\mathcal{M}_1 \leq_H \mathcal{M}_2$ holds for some simulation H.

Lemma 4. *Let $\mathcal{M}_1, \mathcal{M}_2$ be two Kripke structures such that $\mathcal{M}_1 \leq \mathcal{M}_2$. Then $\mathcal{M}_1 \models \mathcal{M}_2$.*

Lemma 4 is a well-known property of simulation. Next, we describe how to construct sequences of over- and under-approximations for a given model $\mathcal{M}$. Each approximation in these sequences is closer to the original model than its previous. We later incorporate these sequences in a MultiLTL abstraction-refinement based model-checking algorithm using our proof rules.

5.1 Constructing a Sequence of Over-Approximations

Given a Kripke structure $\mathcal{M} = (S, I, R, L)$ over AP, we construct an over-approximations sequence $\mathcal{A}_0 \geq \mathcal{A}_1 \geq \cdots \mathcal{A}_k \geq \mathcal{M}$, where $\mathcal{A}_{i+1}$ is a *refinement* of $\mathcal{A}_i$, which we compute by using counterexamples. A *counterexample* is a word $w \in \mathcal{L}(\mathcal{A}_i)$ yet $w \notin \mathcal{L}(\mathcal{M})$. By Lemma 1, it suffices to consider finite prefixes of w, since there is an index j for which $w_0, w_1, \ldots, w_{j-1} \in \mathcal{L}(\mathcal{A}_i) \setminus \mathcal{L}(\mathcal{M})$.

We use a sequence of *abstraction functions* $h_0, \ldots, h_k$, each defining an abstract model.

Definition 5. *Let $\hat{S}$ be a finite set of abstract states. A function $h : S \to \hat{S}$ is an* abstraction function *if h is onto, and for every $\hat{s} \in \hat{S}$, it holds that $L(s_1) = L(s_2)$ for every $s_1, s_2 \in h^{-1}(\hat{s})$.*

Definition 6. *For an abstraction function $h : S \to \hat{S}$, the* $\exists\exists$ abstract model induced by *h is $\mathcal{A}_h = (\hat{S}, \hat{I}, \hat{R}, \hat{L})$, where $\hat{I} = \{\hat{s} \mid \exists s_0 \in I, h(s_0) = \hat{s}\}$, where for every $\hat{s} \in \hat{S}$ we set $\hat{L}(\hat{s}) = L(s)$ for some s such that $h(s) = \hat{s}$, and $(\hat{s}, \hat{s}') \in \hat{R}$ iff there exist $s, s' \in S$ such that $(s, s') \in R$, $h(s) = \hat{s}$ and $h(s') = \hat{s}'$.*[3]

[3] $\hat{L}$ is well defined since by Definition 5, only equilabeled states are mapped to the same abstract state.

Lemma 5. *Let $\mathcal{M} = (S, I, R, L)$ be a Kripke structure and $\mathcal{A}_h = (\hat{S}, \hat{I}, \hat{R}, \hat{L})$ be the $\exists\exists$ abstract model induced by an abstraction function $h : S \to \hat{S}$. Then, $\mathcal{M} \leq \mathcal{A}_h$.*

Proof. The relation $H = \{(s, h(s)) | s \in S\}$ is a simulation from $\mathcal{M}$ to $\mathcal{A}_h$. □

Definition 7. *Let $\mathcal{M}$ and $\mathcal{M}'$ be Kripke structures such that $\mathcal{M} \leq \mathcal{M}'$ by a simulation H, and let $r' = s'_0, s'_1, \ldots$ be a run of $\mathcal{M}'$ on w. The run $r = s_0, s_1, \ldots, s_j$ is a* maximal induced run *of r' in $\mathcal{M}$, if for every $i \in [0, j]$ it holds that $(s_i, s'_i) \in H$, and for every $i \in [0, j-1]$ it holds that $(s_i, s_{i+1}) \in R$. Moreover, there is no state $s^* \in S$ such that $(s^*, s'_{j+1}) \in H$ and $(s_j, s^*) \in R$. If no such j exists then r is infinite, and for every $i \geq 0$ it holds that $(s_i, s'_i) \in H$ and $(s_i, s_{i+1}) \in R$.*

In the sequel, we fix a Kripke structure $\mathcal{M} = (S, I, R, L)$.

Over-Approximation Sequence Construction

Initialization. Define $\hat{S}_0 = \{s_P \mid P \subseteq AP$ and $\exists s \in S : \; L(s) = P\}$. That is, there is a state in $\hat{S}_0$ for every labeling in $\mathcal{M}$. The initial over-approximation $\mathcal{A}_0$ is the $\exists\exists$ model induced by $h_0 : S \to \hat{S}_0$ defined by $h_0(s) = s_{L(s)}$. Since h_0 is an abstraction function, by Lemma 5 we have that $\mathcal{M} \leq \mathcal{A}_0$.

Refinement. Let $h_i : S \to \hat{S}_i$ be an abstraction function. Let $\mathcal{A}_i = (\hat{S}_i, \hat{I}_i, \hat{R}_i, \hat{L}_i)$ be the $\exists\exists$ model induced by h_i. By Lemma 5 we have that $\mathcal{M} \leq \mathcal{A}_i$. Let $w \in \mathcal{L}(\mathcal{A}_i) \backslash \mathcal{L}(\mathcal{M})$ be a counterexample. Let $\hat{r}_i = \hat{s}_0, \hat{s}_1 \ldots$ be a run of $\mathcal{A}_i$ on w, and $r = s_0 \ldots, s_j$ be a maximal induced run of $\mathcal{M}$ on w. Since $w \notin \mathcal{L}(\mathcal{M})$, we have that r is finite. We define $\mathcal{A}_{i+1}$ to be the $\exists\exists$ model induced by h_{i+1}, where $h_{i+1} : S \to \hat{S}_{i+1}$ for $\hat{S}_{i+1} = \hat{S}_i \uplus \{\hat{s}'\}$, defined as follows, for every $s \in S$.

$$h_{i+1}(s) = \begin{cases} h_i(s), & \text{if } h_i(s) \neq \hat{s}_j \\ h_i(s), & \text{if } h_i(s) = \hat{s}_j \text{ and } \exists s' \in S \text{ such that } h_i(s') = \hat{s}_{j+1} \text{ and } (s, s') \in R \\ \hat{s}', & \text{if } h_i(s) = \hat{s}_j \text{ and } \neg\exists s' \in S \text{ such that } h_i(s') = \hat{s}_{j+1} \text{ and } (s, s') \in R \end{cases}$$

The intuition for the refinement is presented in Fig. 1 (a). Concrete states are the full circles and abstract states are the dashed ovals. The purple line is a maximal induced run of $\hat{s}_0, \hat{s}_1 \ldots$ in $\mathcal{M}$, which ends at $\hat{s}_j$. Since there is an infinite run in the abstract model, we can split $\hat{s}_j$ into two abstract states: one that includes all states that can continue to $\hat{s}_{j+1}$, and another that includes all the states with no such transitions. Clearly, the former set includes only states that are not reachable by the maximal induced run of $\hat{s}_0, \hat{s}_1 \ldots$, else the induced run would not have been maximal.

Lemma 6. *For every $i \in \mathbb{N}$, for every state $\hat{s} \in \hat{S}_i$, there exists a state $s \in S$ such that $h_i(s) = \hat{s}$.*

Proof. By induction on i. **Base:** By construction, for every $\hat{s}_P \in \hat{S}_0$ there exists a state $s \in S$ such that $L(s) = P$, and so $h_0(s) = \hat{s}_P$.

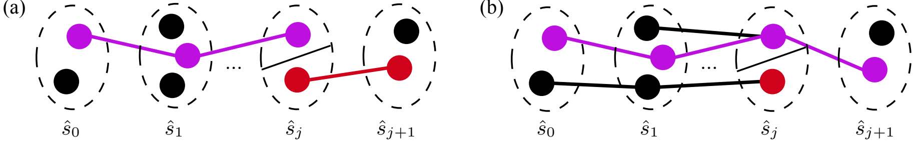

Fig. 1. Refinements (a) $\exists\exists$, and (b) $\forall\exists$.

Step: Assume towards contradiction that there is an abstract state $\hat{s} \in \hat{S}_{i+1}$ such that for every $s \in S$, it holds that $h_{i+1}(s) \neq \hat{s}$. Since $\mathcal{A}_i$ fulfills the required property, $\hat{s} \notin \hat{S}_i$. Then $\hat{s}$ is the new state $\hat{s}'$. Let $s_0, \ldots, s_j$ be a maximal induced run of $\mathcal{M}$ on the counterexample w. There is no state $s' \in h_i^{-1}(\hat{s}_{j+1})$ such that $(s_j, s') \in R$. Thus, by construction, $h_{i+1}(s_j) = \hat{s}'$, a contradiction. $\square$

Lemma 7. *For every $i \geq 0$, it holds that $\mathcal{M} \leq \mathcal{A}_{i+1} \leq \mathcal{A}_i$*

Proof. According to Lemma 5, it is left to show is that $\mathcal{A}_{i+1} \leq \mathcal{A}_i$. The relation $H \subseteq \hat{S}_{i+1} \times \hat{S}_i$, defined by $H = \{(\hat{s}, \hat{s}') \mid h_{i+1}^{-1}(\hat{s}) \subseteq h_i^{-1}(\hat{s}')\}$ is a simulation from $\mathcal{A}_{i+1}$ to $\mathcal{A}_i$. $\square$

Following Lemma 7, we have that $\mathcal{M} \leq \cdots \leq \mathcal{A}_1 \leq \mathcal{A}_0$. Thus, the refinements get more precise with every refinement step. Moreover, for $i > 0$, the model $\mathcal{A}_i$ is obtained from $\mathcal{A}_{i-1}$ by splitting a state. In a finite-state setting, this guarantees termination at the latest when reaching $\mathcal{A}_i = \mathcal{M}$.

Lemma 8. *Let $\mathcal{M}$ be a Kripke structure and let $\mathcal{A}_0 \geq \mathcal{A}_1 \cdots \geq \mathcal{M}$ be our sequence of over-approximations. Then, there exists $m \in \mathbb{N}$ for which $\mathcal{A}_m = \mathcal{A}_{m+1}$.*

5.2 Constructing a Sequence of Under-Approximations

Given $\mathcal{M} = (S, I, R, L)$ over AP, we construct a sequence of under-approximations $\mathcal{A}_0 \leq \mathcal{A}_1 \leq \cdots \leq \mathcal{A}_k \leq \mathcal{M}$ via a sequence of abstraction functions using counterexamples. In this case, a counterexample is a word $w \notin \mathcal{L}(\mathcal{A})$, yet $w \in \mathcal{L}(\mathcal{M})$. Again, we can consider a prefix of w.

Definition 8. *Given an abstraction function $h : S \rightarrow \hat{S}$, the* $\forall\exists$ abstract model *induced by h is $\mathcal{A}_h = (\hat{S}, \hat{I}, \hat{R}, \hat{L})$, where $\hat{I}$ and L are as in Definition 6, and $(\hat{s}, \hat{s}') \in \hat{R}$ iff for every $s \in S$ such that $h(s) = \hat{s}$ there exists $s' \in S$ such that $(s, s') \in R$ and $h(s') = \hat{s}'$.*

Notice that the transition relation $\hat{R}$ of the $\forall\exists$ abstract model might not be total, i.e., there may exist a state with no outgoing transitions.

Lemma 9. *Let $\mathcal{M} = (S, I, R, L)$ be a Kripke structure and $\mathcal{A}_h = (\hat{S}, \hat{I}, \hat{R}, \hat{L})$ be the $\forall\exists$ abstract model induced by an abstraction function $h : S \rightarrow \hat{S}$. Then, $\mathcal{A}_h \leq \mathcal{M}$.*

Proof. $H = \{(h(s), s) | s \in S\}$ is a simulation from $\mathcal{A}_h$ to $\mathcal{M}$. □

Under-approximation Sequence Construction
Initialization. Let $\hat{S}_0$ and h_0 be as in Sect. 5. We set the initial under-approximation $\mathcal{A}_0$ of $\mathcal{M}$ to be the $\forall\exists$ abstract model induced by h_0. By Lemma 9, we have $\mathcal{A}_0 \leq \mathcal{M}$.

Refinement. Let $\mathcal{A}_i = (\hat{S}_i, \hat{I}_i, \hat{R}_i, \hat{L}_i)$ be an $\forall\exists$ abstract model induced by an abstraction function $h_i : S \to \hat{S}_i$. Recall that $\mathcal{A}_i \leq \mathcal{M}$. Let $w \in \mathcal{L}(\mathcal{M}) \setminus \mathcal{L}(\mathcal{A}_i)$ be a counterexample. Let $r = s_0, s_1, \dots$ be a run of $\mathcal{M}$ on w, and let $\hat{r} = \hat{s}_0, \dots, \hat{s}_j$ be a maximal induced run of $\mathcal{A}_i$ on w. We define $\mathcal{A}_{i+1}$ to be the $\forall\exists$ abstract model induced by $h_{i+1} : S \to \hat{S}_{i+1}$ where $\hat{S}_{i+1} = \hat{S}_i \uplus \{\hat{s}'\}$, and where:

$$h_{i+1}(s) = \begin{cases} h_i(s), & \text{if } h_i(s) \neq \hat{s}_j \\ h_i(s), & \text{if } h_i(s) = \hat{s}_j \text{ and } \exists s' \in S \text{ such that } h_i(s') = h_i(s_{j+1}) \text{ and } (s, s') \in R \\ \hat{s}', & \text{if } h_i(s) = \hat{s}_j \text{ and } \neg\exists s' \in S \text{ such that } h_i(s') = h_i(s_{j+1}) \text{ and } (s, s') \in R \end{cases}$$

The idea behind this refinement is represented in Fig. 1 (b). The purple states and lines represent the run in $\mathcal{M}$. Note that in $\hat{s}_j$ there is a red state with no transition to states in $h_i(s_{j+1})$. Thus there is no $\forall\exists$ abstract transition from $\hat{s}_j$ to $h_i(s_{j+1})$. To add such a transition, we split $\hat{s}_j$ into two states: one with all states that have a transition to a state in $h_i(s_{j+1})$, and another with all states that have no such transition. As a result, $\mathcal{A}_{i+1}$ includes a $\forall\exists$ transition from $\hat{s}_j$ to $h_i(s_{j+1})$.

Similarly to over-approximation, we have the following, which assures correctness and termination.

Lemma 10. *Let $\mathcal{M}$ be a model and let $\mathcal{A}_0, \mathcal{A}_1, \dots$ be the sequence of under-approximations described above. Then, the following holds.*

- $\mathcal{A}_0 \leq \mathcal{A}_1 \leq \cdots \leq \mathcal{M}$.
- *There exists $m \in \mathbb{N}$ such that $\mathcal{A}_m = \mathcal{A}_{m+1}$.*

5.3 Abstraction-Refinement Guided MultiLTL Model-Checking Using (PR) and ($\overline{\text{PR}}$)

Following Sects. 5.1 and 5.2, we present an abstraction-refinement inspired approach for model-checking multi-properties. We are given a MultiLTL formula $\mathbb{P} = \mathbb{Q}_1^1\pi_1 \dots \mathbb{Q}_n^n\pi_n\ \varphi$ and a multi-model $\mathbb{M} = \langle \mathcal{M}_1, \dots, \mathcal{M}_n \rangle$ over AP (see footnote 2). The model-checking procedure for $\mathbb{M} \models \mathbb{P}$ is described in Algorithm 1, which we detail next.

The procedure MMC($\mathbb{M}, \mathbb{P}$) performs model-checking as per Lemma 2 (1) and returns **true** if $\mathbb{M} \models \mathbb{P}$, and **false** otherwise. REFINE refines every approximation $\mathcal{A}_i$ for which there is a counterexample w_i in the vector $\langle w_1, \dots, w_n \rangle$ of counterexamples.

Algorithm 1: Abstraction-refinement based MultiLTL model-checking

```
   Input: 𝕄 = ⟨𝓜_1, ..., 𝓜_n⟩, ℙ = ℚ_1^1 π_1 ... ℚ_n^n π_n φ(π_1, ..., π_n).
   Output: 𝕄 ⊨ ℙ?
 1 𝔸, 𝔹 = INITIALIZE(𝕄, ℙ)
 2 while true do
 3 |   res = MMC(𝔸, ℙ)
 4 |   if res == true then
 5 |   |   return 𝕄 ⊨ ℙ
 6 |   else
 7 |   |   ⟨w_1, ..., w_n⟩ = GET_CEX(𝔸, 𝕄, PR)
 8 |   |   𝔸 = REFINE(⟨w_1, ..., w_n⟩, 𝔸)
 9 |   res = MMC(𝔹, ¬ℙ)
10 |   if res == true then
11 |   |   return 𝕄 ⊭ ℙ
12 |   else
13 |   |   ⟨w_1, ..., w_n⟩ = GET_CEX(𝔹, 𝕄, \overline{PR})
14 |   |   𝔹 = REFINE(⟨w_1, ..., w_n⟩, 𝔹)
15 endwhile
```

Initialization. In INITIALIZE (Line 1), for every model $\mathcal{M}_i$ such that $\mathbb{Q}_i^i = \forall$, we initialize abstract models $\mathcal{A}_i$ and $\mathcal{B}_i$ as described Sects. 5.1 and 5.2, respectively. For every model $\mathcal{M}_i$ such that $\mathbb{Q}_i^i = \exists$, we initialize abstract models $\mathcal{A}_i$ and $\mathcal{B}_i$ as described in Sects. 5.2 and 5.1, respectively. Thus, $\mathcal{B}_i \leq \mathcal{M}_i \leq \mathcal{A}_i$ for every $i \in I_\forall$ and $\mathcal{A}_i \leq \mathcal{M}_i \leq \mathcal{B}_i$ for every $i \in I_\exists$. In Algorithm 1, $\mathbb{A} = \langle \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ is used for (PR) and $\mathbb{B} = \langle \mathcal{B}_1, \ldots \mathcal{B}_n \rangle$ for ($\overline{\text{PR}}$).

Abstraction-Refinement. Lines 3–8 apply the rule (PR). When reaching line 3, it is guaranteed that $\mathcal{M}_i \leq \mathcal{A}_i$ for every $i \in I_\forall$ and $\mathcal{A}_i \leq \mathcal{M}_i$ for every $i \in I_\exists$. Thus, we try to apply (PR). We model-check $\mathbb{A} \models \mathbb{P}$ (Line 3). If the result is **true**, then by the correctness of (PR), we have $\mathbb{M} \models \mathbb{P}$ (Line 5). Otherwise, $\mathbb{A} \not\models \mathbb{P}$. As noted in Note 1, for $\mathcal{A}_i$ where $i \in I_\exists$, no single word counterexample can be obtained from the model-checking. Instead, we call GET_CEX (Line 7), which returns a sequence of words that lead to more precise abstractions. For (PR), GET_CEX returns an arbitrary $w_i \in \mathcal{L}(\mathcal{A}_i) \setminus \mathcal{L}(\mathcal{M}_i)$ for every $i \in I_\forall$ and an arbitrary $w_i \in \mathcal{L}(\mathcal{M}_i) \setminus \mathcal{L}(\mathcal{A}_i)$ for every $i \in I_\exists$. For ($\overline{\text{PR}}$), GET_CEX behaves dually on $\mathbb{B}$ for $I_\forall$ and $I_\exists$. If for some i such a word w_i does not exist, GET_CEX returns **null** as w_i. REFINE uses $\langle w_1, \ldots, w_n \rangle$ to refine each abstraction in $\mathbb{A}$ as described in Sects. 5.1, 5.2, obtaining closer abstractions to the original models.

Lines 9–14 apply the rule ($\overline{\text{PR}}$). When we reach line 9, it is guaranteed that $\mathcal{B}_i \leq \mathcal{M}_i$ for every $i \in I_\forall$ and $\mathcal{M}_i \leq \mathcal{B}_i$ for every $i \in I_\exists$. Thus, we try to apply ($\overline{\text{PR}}$) in a similar manner as before. We model-check $\mathbb{B} \models \neg\mathbb{P}$. If the result is **true**, then by the correctness of ($\overline{\text{PR}}$), we have $\mathbb{M} \models \neg\mathbb{P}$ which implies $\mathbb{M} \not\models \mathbb{P}$. Otherwise, we call GET_CEX (Line 13) and refine $\mathbb{B}$ using $\langle w_1, \ldots, w_n \rangle$ (Line 14).

In the worst case, all approximations converge to their respective models (as per Lemmas 8, 10), upon which no further counterexamples are found. Therefore,

the run is guaranteed to terminate. Of course, the run terminates much earlier in case that appropriate approximations are found. Correctness follows from the correctness of (PR) and ($\overline{\text{PR}}$). Hence, we have the following.

Lemma 11. *Algorithm 1 terminates with the correct result.*

Example. Consider $\mathcal{M}_1, \mathcal{M}_2$ (Fig. 2) and $\mathbb{P} = \forall^1 \pi \exists^2 \tau.\ \mathsf{G}(p_\pi \oplus \mathsf{X} p_\pi \oplus \mathsf{XX} p_\pi) \wedge \mathsf{G}(p_\pi \rightarrow q_\tau)$, where $\oplus$ denotes XOR. For brevity, we ignore $\mathbb{B}$ since $\langle \mathcal{M}_1, \mathcal{M}_2 \rangle \models \mathbb{P}$. When running Algorithm 1 for $\langle \mathcal{M}_1, \mathcal{M}_2 \rangle \models \mathbb{P}$, we first construct $\mathcal{A}_1^0, \mathcal{A}_2^0$ as over- and under-approximations of $\mathcal{M}_1, \mathcal{M}_2$, respectively (Fig. 2). Then, we check whether $\langle \mathcal{A}_1^0, \mathcal{A}_2^0 \rangle \models \mathbb{P}$. This does not hold, and MMC returns counterexamples $\langle \emptyset p \emptyset^\omega, \emptyset q^\omega \rangle$. We refine the abstractions according to these counterexamples.

Next, we find the maximal induced run of $\emptyset p \emptyset^\omega$ in $\mathcal{M}_1$, which is the path $\mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{1}\bot$. Since the path for $\emptyset p \emptyset^\omega$ is $\mathbf{4}, \mathbf{5}, \mathbf{4}, \mathbf{4}^\omega$ in $\mathcal{A}_1^0$, we need to refine the state **4** in $\mathcal{A}_1^0$. By similar analysis of $\emptyset q^\omega$, state **8** is to be split in $\mathcal{A}_2^0$. Thus, we split state **4** from $\mathcal{A}_1^0$ to states **6**,**7** in $\mathcal{A}_1^1$. In $\mathcal{A}_2^0$, we split state **8** to states **9**,**10** in $\mathcal{A}_2^1$. Then, model-checking $\langle \mathcal{A}_1^1, \mathcal{A}_2^1 \rangle \models \mathbb{P}$ passes, and we return $\langle \mathcal{M}_1, \mathcal{M}_2 \rangle \models \mathbb{P}$.

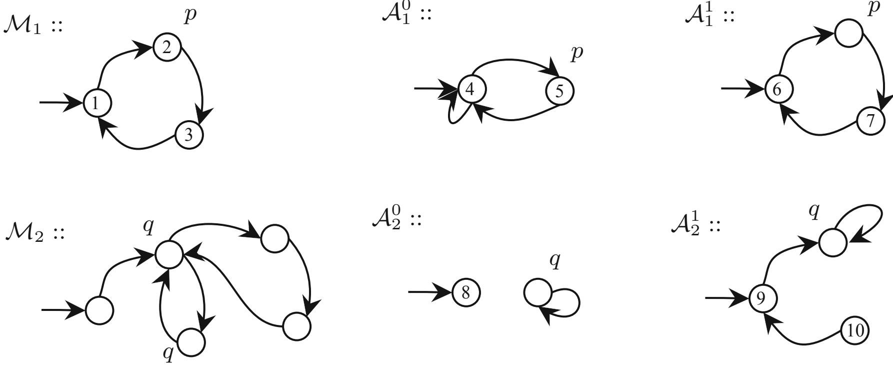

Fig. 2. Model-Checking for $\langle \mathcal{M}_1, \mathcal{M}_2 \rangle \models \mathbb{P}$

5.4 Counterexample Guided MultiLTL Model-Checking Using (PR)

Algorithm 1 is guided by the difference between the abstract models and the original models. We now consider the $\forall^*\exists^*$ fragment of MultiLTL. By Lemma 2, when model-checking $\forall^*\exists^*$MultiLTL fails, we can get counterexamples for the models under $\forall$. We use these counterexamples to further improve our model-checking scheme for this fragment.

We are given a $\forall^*\exists^*$MultiLTL formula $\mathbb{P} = \forall_1^1 \pi_1 \ldots \forall_k^k \pi_k \exists_{k+1}^{k+1} \ldots \exists_n^n \pi_n\ \varphi$ and a multi-model $\mathbb{M} = \langle \mathcal{M}_1, \ldots, \mathcal{M}_n \rangle$ over AP as input. Our model-checking procedure is described in Algorithm 2.

Algorithm 2: CEGAR-based $\forall^*\exists^*$MultiLTL model-checking

Input: $\mathbb{M} = \langle \mathcal{M}_1, \ldots, \mathcal{M}_n \rangle$, $\mathbb{P} = \forall_1^1 \pi_1 \ldots \forall_k^k \pi_k \exists_{k+1}^{k+1} \ldots \exists_n^n \pi_n \; \varphi(\pi_1, \ldots, \pi_n)$.
Output: $\mathbb{M} \models \mathbb{P}$?

```
1  𝔸 = INITIALIZE∀*∃*(𝕄, ℙ)
2  while true do
3  |  (res, cex) = MMC(𝔸, ℙ)
4  |  if res == true then
5  |  |  return 𝕄 ⊨ ℙ
6  |  spuriousList = SPURIOUS(cex, 𝕄)
7  |  if ISEMPTY(spuriousList) then
8  |  |  return 𝕄 ⊭ ℙ
9  |  𝔸 = REFINE(cex, spuriousList, 𝔸, 𝕄)
10 endwhile
```

The procedure MMC($\mathbb{M}, \mathbb{P}$) performs multi-property model-checking, and returns $(true, \emptyset)$ if $\mathbb{M} \models \mathbb{P}$, and otherwise returns $(false, cex)$, where cex is a counterexample vector $\langle w_1, \ldots, w_k \rangle$ such that $w_i \in \mathcal{L}(\mathcal{M}_i)$ for every $i \in [1, k]$ and there are no $w_i \in \mathcal{L}(\mathcal{M}_i)$ for $i \in [k+1, n]$ such that $\langle w_1, \ldots, w_n \rangle \models \mathbb{P}$, as per Lemma 2 (2). We fix every $\mathcal{A}_i$ under $\exists$ to be $\mathcal{M}_i$. Thus, it is guaranteed that the model-checking failure is not caused by words that are missing from the under-approximations, yet do exist in the concrete models. A counterexample w_i from $\langle w_1, \ldots, w_k \rangle$ is *spurious* if $w_i \in \mathcal{L}(\mathcal{A}_i)$ yet $w_i \notin \mathcal{L}(\mathcal{M}_i)$. That is, w_i cannot serve as proof that $\mathbb{M} \not\models \mathbb{P}$. REFINE refines every approximation $\mathcal{A}_i$ for which there is a tuple (i, w_i) in *spuriousList*, the list of spurious counterexamples, by removing w_i from $\mathcal{A}_i$.

Initialization. In INITIALIZE$\forall^*\exists^*$ (Line 1), for every model $\mathcal{M}_i$ such that $\mathbb{Q}_i^i = \forall$, we initialize an abstract model $\mathcal{A}_i$ as described in 5.1. For every model $\mathcal{M}_i$ such that $\mathbb{Q}_i^i = \exists$, we fix $\mathcal{A}_i$ to be $\mathcal{M}_i$. Thus, $\mathcal{M}_i \leq \mathcal{A}_i$ for every $i \in [1, k]$ and $\mathcal{A}_i \leq \mathcal{M}_i$ for every $i \in [k+1, n]$.

Model-Checking. When we reach line 3, it is guaranteed that $\mathcal{M}_i \leq \mathcal{A}_i$ for every $i \in I_\forall$ (and $\mathcal{A}_i \leq \mathcal{M}_i$ for every $i \in I_\exists$, since $\mathcal{A}_i = \mathcal{M}_i$). Thus, we try to apply the proof rule (PR), and model-check $\langle \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle \models \mathbb{P}$ (Line 3) by running MMC. If the result is **true**, then by (PR), we have $\mathbb{M} \models \mathbb{P}$ (Line 5). Otherwise, we get a counterexample vector of the form $\langle w_1, \ldots, w_k \rangle$.

Counterexample Analysis. (Lines 6–9). The procedure SPURIOUS iterates over the words in the counterexample $\langle w_1, \ldots, w_k \rangle$, and returns a list of tuples (i, w_i) such that $w_i \notin \mathcal{L}(\mathcal{M}_i)$. Note that since $\langle w_1, \ldots, w_k \rangle$ is a counterexample, it holds that $w_i \in \mathcal{L}(\mathcal{A}_i)$ for every $i \in [1, k]$. Thus, every w_i in the list of (i, w_i) is spurious. If there are no spurious counterexamples, then we return $\mathbb{M} \not\models \mathbb{P}$ (Line 8). Otherwise, we refine the approximations based on the spurious counterexamples.

In the worst case, the run iterates until $\mathcal{M}_i = \mathcal{A}_i$ for every $i \in [1, n]$, in which case there are no spurious counterexamples. Of course, termination may happen

much earlier. Correctness follows from the correctness of (PR). Hence, we have the following.

Lemma 12. *Algorithm 2 terminates with the correct result.*

Algorithm 2 improves Algorithm 1 in several ways. First, in order to compute the counterexamples there is no need to complement the models, which comes with an exponential price. Second, the counterexamples are provided by the model-checking process. As such, they are of "higher quality", in the sense that they take into account the checked property and are guaranteed to remove refuting parts from the abstractions. This, in turn, leads to faster convergence.

6 Multi-properties for Finite Traces

We now consider models whose traces are finite. This setting is natural, for example, when modeling terminating programs. In this case, a model is a finite-word language, and hyperproperties can be expressed by nondeterminisitic finite hyperautomata (NFH) [7]. To explain the idea behind NFH, we first review nondeterministic automata.

Definition 9. *A* nondeterministic finite-word automaton *(NFA) is a tuple* $A = (\Sigma, Q, Q_0, \delta, F)$*, where* Σ *is an alphabet,* Q *is a nonempty finite set of* states*,* $Q_0 \subseteq Q$ *is a set of* initial states*,* $F \subseteq Q$ *is a set of* accepting states*, and* $\delta \subseteq Q \times \Sigma \times Q$ *is a* transition relation*.*

Given a word $w = \sigma_1\sigma_2\cdots\sigma_n$ over Σ, a *run of* A *on* w is a sequence of states $(q_0, q_1, \ldots q_n)$, such that $q_0 \in Q_0$, and for every $0 < i \leq n$, it holds that $(q_{i-1}, \sigma_i, q_i) \in \delta$. The run is *accepting* if $q_n \in F$. The *language* of A, denoted $\mathcal{L}(A)$, is the set of all words on which A has an accepting run. A language $\mathcal{L}$ is called *regular* if there exists an NFA such that $\mathcal{L}(A) = \mathcal{L}$.

An NFA A is called *deterministic* (DFA) if $|Q_0| = 1$, and for every $q \in Q$ and $\sigma \in \Sigma$, there exists exactly one q' for which $(q, \sigma, q') \in \delta$. It is well-known that every NFA has an equivalent DFA.

We now turn to explain NFH. An NFH $\mathcal{A}$ consists of a set of *word variables*, an NFA $\mathsf{nfa}(\mathcal{A})$ that runs on words that are assigned to these variables (which is akin to the unquantified LTL formula in a HyperLTL formula), and a *quantification condition* that describes the requirements for these assignments (which is akin to the quantifiers in a HyperLTL formula). Thus, NFH can be thought of as the regular-language counterpart of HyperLTL. We demonstrate NFH with an example.

Example 4. Consider the NFH $\mathcal{A}$ in Fig. 3 (left) over the alphabet $\Sigma = \{a, b\}$ and two word variables x and y. The NFA part $\mathsf{nfa}(\mathcal{A})$ of $\mathcal{A}$ reads two words simultaneously: one is assigned to x and the other to y. Accordingly, the letters that $\mathsf{nfa}(\mathcal{A})$ reads are tuples of the form $\{\sigma_x, \sigma'_y\}$, where σ is the current letter in the word that is assigned to x, and similarly for σ' and y. The symbol # is used for padding at the end if one of the words is shorter than the other. In

the example, for two words w_1, w_2 that are assigned to x and y, respectively, $\mathsf{nfa}(\mathcal{A})$ requires that (1) w_1, w_2 agree on their a positions, and (2) once one of the words has ended, the other must only contain b letters. Since the quantification condition of $\mathcal{A}$ is $\forall x \forall y$, in a language S that $\mathcal{A}$ accepts, every two words agree on their a positions. As a result, all the words in S must agree on their a positions. The *hyperlanguage* of $\mathcal{A}$ is then the set of all finite-word languages in which all words agree on their a positions.

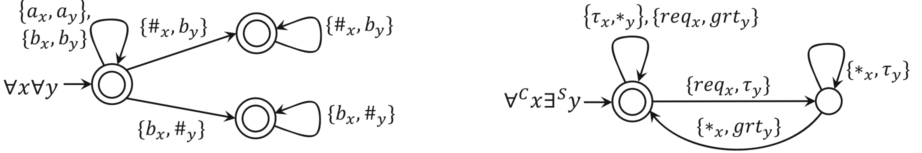

Fig. 3. The NFH $\mathcal{A}$ (left) and the MNFH $\mathcal{B}$ (right).

The *model-checking problem for NFH* is to decide, given a language S and an NFH $\mathcal{A}$, whether $\mathcal{A}$ accepts S, in which case we denote $S \models \mathcal{A}$. When S is given as an NFA, the model-checking problem is decidable (albeit, as for HyperLTL, by a nonelementary algorithm) [7].

6.1 Multi-languages and Multi-NFH

As in the case of models with infinite traces, we generalize languages and NFH to *multi-languages* and *multi-NFH* (MNFH). Thus, a multi-language is a tuple $\langle S_1, S_2, \ldots S_k \rangle$ of finite-word languages, and an MNFH $\mathcal{A}$ is an NFH with indexed quantifiers. The semantics is similar to that of Sect. 3, i.e., a quantifier $\mathbb{Q}^i$ in the quantification condition of $\mathcal{A}$ refers to S_i (rather than all quantifiers referring to the same language in the case of standard NFH).

We consider multi-languages that consist of regular languages. We can express such a multi-language $\langle L_1, L_2, \ldots, L_k \rangle$ by a tuple $\mathbb{M} = \langle \mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_k \rangle$ of NFAs, where $\mathcal{L}(\mathcal{M}_i) = L_i$ for every $i \in [1, k]$. We call $\mathbb{M}$ a *multi-NFA* (MNFA). We define the model-checking problem for MNFA accordingly, and denote $\mathbb{M} \models \mathbb{P}$ if an MNFH $\mathbb{P}$ accepts $\mathbb{M}$.

Example 5. Consider an MNFA $\langle S, C \rangle$, where S models a server and C models a client, and the MNFH $\mathcal{B}$ of Fig. 3 (right) over $\Sigma = \{req, grt, \tau\}$, where req is a request sent to the server, grt is a grant given to the client and τ is a non-communicating action.

The multi-model $\langle S, C \rangle$ satisfies $\mathcal{B}$ iff for every run of C there exists a run of S such that every request by C is eventually granted by S. This means that the server does not starve the client.

From now on, we assume without loss of generality that the quantification conditions of the MNFH that we consider are of the form $\mathbb{Q}_1^1 x_1 \mathbb{Q}_2^2 x_2 \ldots \mathbb{Q}_k^k x_k$.

We now show that the model-checking problem for MNFH is equivalent to the model-checking problem for NFH. For the first direction, it is easy to see that a language S is accepted by an NFH $\mathcal{A}$ iff $\langle S \rangle$ is accepted by the MNFH $\mathcal{A}'$ obtained from $\mathcal{A}$ by indexing all quantifiers in the quantification condition of $\mathcal{A}$ by the same index 1. We now show the second direction.

Theorem 2. *Let $\mathbb{P}$ be an MNFH, and let $\mathbb{M} = \langle \mathcal{M}_1, \ldots \mathcal{M}_n \rangle$ be an MNFA. Then there exist an NFA $\mathcal{M}$ and an NFH $\mathcal{P}$ such that $\mathbb{M} \models \mathbb{P}$ iff $\mathcal{M} \models \mathcal{P}$.*

Proof Sketch. We first mark the individual traces of every NFA in $\mathbb{M}$ by adding its index to all its letters. That is, we replace every letter σ in $\mathcal{M}_i$ with (σ, i). Then, we union all the NFAs in the updated MNFA $\mathbb{M}$ to a single NFA $\mathcal{M}$. Now, every word w in $\mathcal{L}(\mathcal{M})$ is marked with the index of the NFA in $\mathbb{M}$ from which it originated.

We translate the MNFH $\mathbb{P}$ to an NFH $\mathcal{P}$ as follows. First, we remove the indices from the quantifiers in the quantification condition α of $\mathbb{P}$. Next, recall that a letter in $\mathsf{nfa}(\mathbb{P})$ is in fact a letter-set of the form $\{\sigma_{1_{x_1}}, \ldots \sigma_{k_{x_k}}\}$. We update these letters according to $\mathcal{M}$: for every variable x, if x is under the quantifier $\mathbb{Q}^i x$ in α, then we replace every occurrence of σ_x in $\mathsf{nfa}(\mathbb{P})$ with $(\sigma, i)_x$.

Every $\exists^i x$ in α requires the existence of a word $w \in \mathcal{L}(\mathcal{M}_i)$ that is assigned to x and is accepted by $\mathsf{nfa}(\mathbb{P})$ (along with other words assigned to the other variables). Accordingly, $\mathcal{P}$ now requires the existence of a word $w \in \mathcal{L}(\mathcal{M})$ that originates from $\mathcal{L}(\mathcal{M}_i)$ that is assigned to x and is accepted by $\mathsf{nfa}(\mathcal{P})$. That is, the requirement for $\exists$ quantifiers is maintained.

To maintain the requirements for $\forall$ quantifiers, we add a new accepting sink q to $\mathsf{nfa}(\mathcal{P})$, and add transitions to q from every state with every letter-set in which a letter $(\sigma, j)_x$ occurs, where α includes $\forall^i x$ for $i \neq j$. Intuitively, $\forall$ quantifiers in $\mathcal{P}$ require that every word from $\mathcal{L}(\mathcal{M})$ that is assigned to x is accepted by $\mathsf{nfa}(\mathcal{P})$. Since in $\mathbb{P}$ we only required every word from $\mathcal{L}(\mathcal{M}_i)$ to be accepted, we use q to accept words from all the other NFAs in $\mathbb{M}$ that are assigned to x. $\square$

The construction in the proof of Theorem 2 uses an alphabet whose size is polynomial in the original alphabet. The model $\mathcal{M}$ that we construct is linear in the size of $\mathbb{M}$, and the state space of $\mathcal{P}$ is linear in that of $\mathbb{P}$. However, since the size of the alphabet is larger, and the letters of $\mathcal{P}$ are set-letters, there may be exponentially many transitions in $\mathcal{P}$ compared with $\mathbb{P}$.

However, the model-checking algorithm from [7] can be easily altered to handle MNFH, without going through the reduction. Additionally, when $\mathbb{M} \not\models \mathbb{P}$, it is possible to extract a counterexample $\langle w_1, \ldots, w_k \rangle$ when $\mathbb{Q}_i = \forall$ for $i \in [1, k]$.

Lemma 13. *There is a direct algorithm for model-checking MNFH.*

7 Learning-Based Multi-property Model-Checking

We now describe ways of finding approximations according to the proof rules (PR) and ($\overline{\text{PR}}$) described in Sect. 4, for the multi-models of MNFA and

multi-properties of MNFH of Sect. 6. The correctness of our rules stems only from the semantics of the quantifiers and so still holds.

The L^* algorithm [2] is a learning algorithm that finds a minimal DFA for an unknown regular language U. We exploit the fact that MNFA consist of regular languages to introduce an L^*-based algorithm for constructing approximations for the languages in the MNFA and for model-checking MNFH. To explain the idea behind our method, we first describe the L^* algorithm.

The L^* Algorithm. L^* consists of two entities: a *learner*, whose goal is to construct a DFA for U, and a *teacher*, who helps the learner by answering *membership queries* – "is $w \in U$?", and *equivalence queries* – "is A a DFA for U?". In case that $\mathcal{L}(A) \neq U$, the teacher also returns a counterexample: a word which is accepted by A and is not in U, or vice versa.

The learner maintains an *observation table* T that contains words for which a membership query was issued, along with the answers the teacher returned for these queries. Once T fulfills certain conditions (in which case we say that T is *steady*), it can be translated to a DFA A_T whose language is consistent with T. If $\mathcal{L}(A_T) = U$ then L^* terminates. Otherwise, the teacher returns a counterexample with which the learner updates T, and the run continues.

In each iteration, the learner is guaranteed to steady T, and L^* is guaranteed to terminate successfully. The sizes of the DFAs that the learner produces grow from one equivalence query to the next (while never passing the minimal DFA for U). The runtime of L^* is polynomial in the size of a minimal DFA for U and in the length of the longest counterexample that is returned by the teacher.

The main idea behind learning-based model-checking algorithms is to use the candidates produced by the learner as potential approximations. Since these candidates may be significantly smaller than the original models, model-checking is accelerated.

We first introduce our algorithm for the general case, in which L^* aims to learn the models themselves. Then, we introduce an improved algorithm in case that the quantification condition is of the type $\forall\exists$, in which case we can both define stronger learning goals, and use the counterexamples provided by the model-checker to reach these goals more efficiently.

7.1 Learning Assumptions for General Multi-properties

Consider an MNFA $\mathbb{M} = \langle \mathcal{M}_1, \mathcal{M}_2, \ldots \mathcal{M}_k \rangle$, and an MNFH $\mathbb{P}$ with a quantification condition $\alpha = \mathbb{Q}_1^1 x_1 \mathbb{Q}_2^2 x_2 \cdots \mathbb{Q}_k^k x_k$. Algorithm L^*_{MNFH}, described in Algorithm 3, computes an over-approximation for every $\mathcal{M}_i$ under $\forall$, and an under-approximation for every $\mathcal{M}_i$ under $\exists$. It does so by running L^* for every $\mathcal{M}_i$ in parallel, aiming to learn $\mathcal{M}_i$. Thus, the learner maintains a set $T_1, \ldots T_k$ of observations tables, one for every $\mathcal{M}_i$. Whenever all tables are steady, the learner submits the DFAs $\mathcal{A}_{T_1}, \ldots \mathcal{A}_{T_k}$ that it produces as candidates for the approximations via an equivalence query. The result of the equivalence query either resolves $\mathbb{M} \models \mathbb{P}$ according to (PR) and ($\overline{\text{PR}}$), or returns counterexamples with which the learner updates the tables to construct the next round of candidates.

In Algorithm 3, The methods INITIALIZE and STEADY are learner functions used for initializing an observation table, and reaching a steady observation table, respectively. The method ADDCEX updates the table when a counterexample is returned from an equivalence query.

Handling membership queries is rather straightforward: when the learner submits a query w for an NFA $\mathcal{M}_i$, we return **true** iff $w \in \mathcal{L}(\mathcal{M}_i)$. We now describe how to handle equivalence queries.

Equivalence Queries. The learner submits its candidate $\mathbb{A}$, which includes its set of candidates. We first check that they are approximations for (PR), by checking whether $\mathcal{M}_i \models \mathcal{A}_{T_i}$ for every over-approximation and $\mathcal{A}_{T_i} \models \mathcal{M}_i$ for every under-approximation.

If all checks pass, then we model-check $\mathbb{A} \models \mathbb{P}$. If the check passes, we return $\mathbb{M} \models \mathbb{P}$. If the candidates are not approximations for (PR) but are approximations for ($\overline{\text{PR}}$), we model-check $\mathbb{A} \models \neg\mathbb{P}$. If the check passes, we return $\mathbb{M} \not\models \mathbb{P}$.

If none of the above has triggered a return value, then there exists at least one candidate $\mathcal{A}_i$ such that $\mathcal{L}(\mathcal{A}_i) \neq \mathcal{L}(\mathcal{M}_i)$. We can locate these candidates during the over- and under-approximation checks, while computing a word $w \in \mathcal{L}(\mathcal{M}_i) \setminus \mathcal{L}(\mathcal{A}_i)$ (in case that we found $\mathcal{A}_i$ not to be an over-approximation), or a word $w \in \mathcal{L}(\mathcal{A}_i) \setminus \mathcal{L}(\mathcal{M}_i)$ (in the dual case). We then return the list of counterexamples according to the candidates for which we found a counterexample.

Algorithm 3: L^*_{MNFH}

Input: $\mathbb{M} = \langle \mathcal{M}_1, \ldots, \mathcal{M}_k \rangle$, $\mathbb{P}$ with $\alpha = \mathbb{Q}^1_1 \pi_1 \ldots \mathbb{Q}^k_k \pi_k$.
Output: $\mathbb{M} \models \mathbb{P}$?

```
1  INITIALIZE(T_1, ... T_k)
2  while true do
3      foreach i ∈ [1, k] do
4          T_i = STEADY(T_i)
5          Construct A_{T_i} from T_i
6      𝔸 = ⟨A_{T_1}, A_{T_2}, ... A_{T_k}⟩
7      (CexList, pass) = EQUIV(𝔸, 𝕄, ℙ)
8      if CexList == null then
9          if pass then
10             return 𝕄 ⊨ ℙ
11         else
12             return 𝕄 ⊭ ℙ
13     foreach (w_i, i) ∈ CexList do
14         ADDCEX(T_i, w_i)
15 endwhile
```

Since L^* is guaranteed to terminate when learning a regular language, Algorithm 3 is guaranteed to terminate. The correctness of (PR) and ($\overline{\text{PR}}$) guarantee that L^*_{MNFH} terminates correctly at the latest after learning $\mathbb{M}$ (and terminates earlier if it finds smaller appropriate approximations).

7.2 Weakest Assumption for $\forall\exists$

We introduce a *weakest assumption* in the context of multi-properties with a quantification condition $\forall\exists$. Intuitively, a weakest assumption is the most general language that can serve as an over-approximation. We prove that the weakest assumption is regular, and show how to incorporate it in a learning-based multi-property model-checking algorithm based on (PR).

We denote MNFH with a quantification condition of the form $\forall_1^1 x \exists_2^2 y$ by MNFH$_{\forall\exists}$. The weakest assumption is the goal of the learning Algorithm 4 below.

Definition 10. *Let* $\mathbb{M} = \langle \mathcal{M}_1, \mathcal{M}_2 \rangle$ *be an MNFA and let* $\mathbb{P}$ *be an MNFH*$_{\forall\exists}$. *The* weakest assumption for $\mathbb{P}$ w.r.t. $\mathcal{M}_2$ *is as follows.*

$$W^{\mathcal{M}_2:\mathbb{P}} = \bigcup_{\mathcal{A} \ s.t. \ \langle \mathcal{A}, \mathcal{M}_2 \rangle \models \mathbb{P}} \mathcal{L}(\mathcal{A})$$

That is, $W^{\mathcal{M}_2:\mathbb{P}}$ is the union of all languages that along with $\mathcal{M}_2$ satisfy $\mathbb{P}$.

Lemma 14. *Let* $\mathcal{A}$ *and* $\mathcal{M}_2$ *be NFA, and* $\mathbb{P}$ *be an MNFH*$_{\forall\exists}$. *Then* $\mathcal{L}(\mathcal{A}) \subseteq W^{\mathcal{M}_2:\mathbb{P}}$ *iff* $\langle \mathcal{A}, \mathcal{M}_2 \rangle \models \mathbb{P}$.

Proof. If $\langle \mathcal{A}, \mathcal{M}_2 \rangle \models \mathbb{P}$ then the claim holds by the definition of $W^{\mathcal{M}_2:\mathbb{P}}$.
For the other direction, if $\mathcal{L}(\mathcal{A}) \subseteq W^{\mathcal{M}_2:\mathbb{P}}$, then for every $w \in \mathcal{L}(\mathcal{A})$ there exists an NFA $\mathcal{A}_w$, with $\mathcal{L}(\mathcal{A}_w) = \{w\}$ s.t. $\langle \mathcal{A}_w, \mathcal{M}_2 \rangle \models \mathbb{P}$. Therefore, for every $w \in \mathcal{L}(\mathcal{A})$, there exists a word $w' \in \mathcal{L}(\mathcal{M}_2)$ s.t. $\mathbb{P}$ accepts $\{w_x, w'_y\}$, and so by the semantics of MNFH, we have that $\langle \mathcal{A}, \mathcal{M}_2 \rangle \models \mathbb{P}$. $\square$

We note that a similar approach to Lemma 14 cannot work for general quantification conditions, since their satisfying assignments are generally not closed under union.

To justify using $W^{\mathcal{M}_2:\mathbb{P}}$ as the objective of a learning algorithm, we show that $W^{\mathcal{M}_2:\mathbb{P}}$ is regular.

In the following Lemma, $\mathcal{A}_{\Sigma^*}$ is an NFA that accepts all words over Σ, and $\cap$ denotes the intersection construction for NFA. Also, for NFA $\mathcal{A}$ and $\mathcal{B}$, the NFA $\mathcal{A} \times \mathcal{B}$ denotes the NFA over letters of the type $\{\sigma_x, \sigma'_y\}$ where σ_x is from $\mathcal{A}$ and σ'_y is from $\mathcal{B}$ that is formed by running both NFA in parallel, each with its own word (with # padding the end of the shorter word), and $\downarrow_i$ denotes the projection of the parallel construction to the i'th NFA.

Lemma 15. *Let* $\mathbb{P}$ *be an MNFH*$_{\forall\exists}$ *and let* $\mathbb{M} = \langle \mathcal{M}_1, \mathcal{M}_2 \rangle$ *be an MNFA. Then* $w \in W^{\mathcal{M}_2:\mathbb{P}}$ *iff* $w \in \mathcal{L}((\mathsf{nfa}(\mathbb{P}) \cap (\mathcal{A}_{\Sigma^*} \times \mathcal{M}_2)) \downarrow_1)$.

That is, we can derive $W^{\mathcal{M}_2:\mathbb{P}}$ by taking the lefthand-side projection of the parallel run of $\mathsf{nfa}(\mathbb{P})$ with a multi-language consisting of an NFA that accepts all words in Σ^*, and $\mathcal{M}_2$ (while ignoring the # symbols). Intuitively, this projection includes all the words which can be matched with a word in $\mathcal{M}_2$ in a way that is accepted by $\mathsf{nfa}(\mathbb{P})$. We can therefore deduce the following.

Corollary 1. $W^{\mathcal{M}_2:\mathbb{P}}$ *is regular.*

Algorithm 4: $L^*_{\forall\exists}$

Input: An MNFH$_{\forall\exists}$ $\mathbb{P}$, an MNFA $\mathbb{M} = \langle \mathcal{M}_1, \mathcal{M}_2 \rangle$.
Output: $\mathbb{M} \models \mathbb{P}$?

```
1  Initialize(T)
2  while true do
3      T = steady(T)
4      Construct A_T from T
5      (cex, pass) = Equiv(A_T, M, P)
6      if cex then
7          AddCex(T, cex)
8      else
9          if pass then
10             return ⟨M_1, M_2⟩ ⊨ P
11         else
12             return ⟨M_1, M_2⟩ ⊭ P
13 endwhile
```

7.3 Learning Assumptions for $\forall\exists$

Let $\mathbb{P}$ be an MNFH$_{\forall\exists}$ and let $\mathbb{M} = \langle \mathcal{M}_1, \mathcal{M}_2 \rangle$ be an MNFA. We now introduce our $L^*_{\forall\exists}$ learning-based algorithm for model-checking $\mathbb{M} \models \mathbb{P}$. As we have mentioned in 7.2, the learning goal in our $L^*_{\forall\exists}$ algorithm is $W^{\mathcal{M}_2:\mathbb{P}}$, as it is an over-approximation of $\mathcal{M}_1$ (when $\mathbb{M} \models \mathbb{P}$). However, in this case, notice that every $\mathcal{A}$ such that $\mathcal{L}(\mathcal{M}_1) \subseteq \mathcal{L}(\mathcal{A}) \subseteq W^{\mathcal{M}_2:\mathbb{P}}$ suffices. $L^*_{\forall\exists}$ then runs L^* while using every DFA $\mathcal{A}$ that is produced by the learner during the run as a candidate for an over-approximation of $\mathcal{M}_1$.

We now describe our implementation for answering the membership and equivalence queries.

Membership Queries. When the learner submits a membership query $w \in^? \mathcal{L}(\mathcal{A})$, we model-check $\langle \mathcal{A}_w, \mathcal{M}_2 \rangle \models \mathbb{P}$, where $\mathcal{A}_w$ is a DFA whose language is $\{w\}$. If the check passes, then there exists a word $w' \in \mathcal{L}(\mathcal{M}_2)$ such that $\langle w, w' \rangle \models \mathbb{P}$. Therefore, we return **true**. Otherwise, $\langle w, w' \rangle \not\models \mathbb{P}$ for every $w' \in \mathcal{L}(\mathcal{M}_2)$, and thus we do not include w in $\mathcal{L}(\mathcal{A})$, and return **false**.

Equivalence Queries. We first check that $\mathcal{A}$ is a potential over-approximation, by checking if $\mathcal{M}_1 \models \mathcal{A}$. If not, then we return a counterexample $w \in \mathcal{L}(\mathcal{M}_1) \setminus \mathcal{L}(\mathcal{A})$. Otherwise, we model-check $\langle \mathcal{A}, \mathcal{M}_2 \rangle \models \mathbb{P}$. If the model-checking passed, then we can conclude $\mathbb{M} \models \mathbb{P}$. Otherwise, a counterexample w is returned for a word in $\mathcal{L}(\mathcal{M}_1)$ which has no match in $\mathcal{L}(\mathcal{M}_2)$. We now need to check if w is spurious. If $w \notin \mathcal{L}(\mathcal{M}_1)$, then we return w as a counterexample to the learner. Otherwise, we can conclude that $\mathbb{M} \not\models \mathbb{P}$.

Since L^* is guaranteed to terminate when learning a regular language, $L^*_{\forall\exists}$ is guaranteed to terminate. In both cases, when $\mathbb{M} \models \mathbb{P}$ or $\mathbb{M} \not\models \mathbb{P}$, the correctness of PR and the properties of $W^{\mathcal{M}_2:\mathbb{P}}$ guarantee that the algorithm terminates with a correct answer, at most after learning $W^{\mathcal{M}_2:\mathbb{P}}$ (and may terminate earlier if it finds a smaller appropriate over-approximation).

There are several advantages to using Algorithm 4 over Algorithm 3. First, $W^{\mathcal{M}_2:\mathbb{P}}$ may be smaller than $\mathcal{M}_1$ which leads to quicker convergence. Second, there is no need to complement $\mathcal{M}_1$ for the equivalence query, since we only check if $\mathcal{M}_1$ is contained in the candidate submitted by the learner (which is a DFA and can be easily complemented). Finally, we can now use the more targeted counterexample provided by the model-checking process, again leading to quicker convergence.

While we have defined the weakest assumption and Algorithm 4 for a quantification condition of the type $\forall\exists$, both can be easily extended to handle a sequence of $\exists$ quantifiers rather than a single one.

8 Conclusion

We have introduced multi-models and multi-properties – useful notions that generalize hyperproperties to handle multiple systems. We have formalized these notions for both finite- and infinite-trace systems, and presented compositional proof rules for model-checking multi-properties.

For infinite-trace systems, we have introduced MultiLTL, a generalization of HyperLTL, and have applied our proof rules in abstraction-refinement and CEGAR based algorithms. For finite-trace systems, we have introduced multi-NFH, which offer an automata-based specification formalism for regular multi-properties. Here, we have applied our proof rules in automata-learning algorithms. The algorithms for both approaches accelerate model-checking by computing small abstractions, that allow avoiding model-checking the full multi-model.

References

1. Ábrahám, E., Bonakdarpour, B.: HyperPCTL: a temporal logic for probabilistic hyperproperties. In: McIver, A., Horvath, A. (eds.) QEST 2018. LNCS, vol. 11024, pp. 20–35. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99154-2_2
2. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987)
3. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 200–214. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21437-0_17
4. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004, pp. 100–114. IEEE (2004)
5. Bérard, B., Haar, S., Hélouët, L.: Hyper partial order logic. In: 38th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, p. 1 (2018)
6. Bohrer, B., Platzer, A.: A hybrid, dynamic logic for hybrid-dynamic information flow. In: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, pp. 115–124 (2018)
7. Bonakdarpour, B., Sheinvald, S.: Automata for hyperlanguages. arXiv preprint arXiv:2002.09877 (2020)

8. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: Abadi, M., Kremer, S. (eds.) POST 2014. LNCS, vol. 8414, pp. 265–284. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54792-8_15
9. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. **18**(6), 1157–1210 (2010)
10. Dimitrova, R., Finkbeiner, B., Kovács, M., Rabe, M.N., Seidl, H.: Model checking information flow in reactive systems. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 169–185. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27940-9_12
11. Finkbeiner, B., Haas, L., Torfah, H.: Canonical representations of k-safety hyperproperties. In: 2019 IEEE 32nd Computer Security Foundations Symposium (CSF), pp. 17–1714. IEEE (2019)
12. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for model checking HyperLTL and HyperCTL*. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 30–48. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_3
13. Finkbeiner, B., Zimmermann, M.: The first-order logic of hyperproperties. In: 34th Symposium on Theoretical Aspects of Computer Science (STACS 2017). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2017)
14. Nguyen, L.V., Kapinski, J., Jin, X., Deshmukh, J.V., Johnson, T.T.: Hyperproperties of real-valued signals. In: Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design, pp. 104–113 (2017)
15. Păsăreanu, C.S., Giannakopoulou, D., Bobaru, M.G., Cobleigh, J.M., Barringer, H.: Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning. Form. Meth. Syst. Des. **32**(3), 175–205 (2008)
16. Pucella, R., Schneider, F.B.: Independence from obfuscation: a semantic framework for diversity. J. Comput. Secur. **18**(5), 701–749 (2010). https://doi.org/10.3233/JCS-2009-0379
17. Shemer, R., Gurfinkel, A., Shoham, S., Vizel, Y.: Property directed self composition. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 161–179. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_9
18. Sousa, M., Dillig, I.: Cartesian hoare logic for verifying k-safety properties. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 57–69 (2016)
19. Terauchi, T., Aiken, A.: Secure information flow as a safety problem. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 352–367. Springer, Heidelberg (2005). https://doi.org/10.1007/11547662_24
20. Thomas, W.: Path logics with synchronization. In: Perspectives in Concurrency Theory, pp. 469–481 (2009)
21. Yang, W., Vizel, Y., Subramanyan, P., Gupta, A., Malik, S.: Lazy self-composition for security verification. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 136–156. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_11