Specifying and Testing k-Safety Properties for Machine-Learning Models

Maria Christakis¹, Hasan Ferit Eniser², Jörg Hoffmann³,⁴, Adish Singla² and Valentin Wüstholz⁵

¹TU Wien, Austria ²MPI-SWS, Germany ³Saarland University, Saarland Informatics Campus, Germany ⁴German Research Center for Artificial Intelligence (DFKI), Germany ⁵ConsenSys, Austria

maria.christakis@tuwien.ac.at, {hfeniser, adishs}@mpi-sws.org, hoffmann@cs.uni-saarland.de, valentin.wustholz@consensys.net

Abstract

Machine-learning models are becoming increasingly prevalent in our lives, for instance assisting in image-classification or decision-making tasks. Consequently, the reliability of these models is of critical importance and has resulted in the development of numerous approaches for validating and verifying their robustness and fairness. However, beyond such specific properties, it is challenging to specify, let alone check, general functional-correctness expectations from models. In this paper, we take inspiration from specifications used in formal methods, expressing functional-correctness properties by reasoning about k different executions—so-called k-safety properties. Considering a credit-screening model of a bank, the expected property that "if a person is denied a loan and their income decreases, they should still be denied the loan" is a 2-safety property. Here, we show the wide applicability of k-safety properties for machine-learning models and present the first specification language for expressing them. We also operationalize the language in a framework for automatically validating such properties using metamorphic testing. Our experiments show that our framework is effective in identifying property violations, and that detected bugs could be used to train better models.

1 Introduction

Due to the impressive advances in machine learning and the unlimited availability of data, machine-learning (ML) models, e.g., neural networks, are rapidly becoming prevalent in our lives, for instance by assisting in image-classification or decision-making tasks. As a result, there is growing concern about the reliability of these models in performing such tasks. For example, it could be disastrous if an autonomous vehicle misclassifies a street sign, or if a recidivism-risk algorithm, which predicts whether a criminal is likely to re-offend, is unfair with respect to race. The research community is, of course, aware of these issues and has devised numerous techniques to validate and verify robustness and fairness properties of machine-learning models (e.g., [Huang et al., 2017; Gehr et al., 2018; Singh et al., 2019; Albarghouthi et al., 2017; Bastani et al., 2019; Urban et al., 2020; Carlini and Wagner, 2017; Goodfellow et al., 2015; Madry et al., 2018; Galhotra et al., 2017; Udeshi et al., 2018; Tramèr et al., 2017]).

Beyond such specific properties however, it is challenging to express general functional-correctness expectations from such models, let alone check them, e.g., how can we specify that an image classifier should label images correctly? We take inspiration from specifications used in formal methods—so-called hyperproperties [Clarkson and Schneider, 2008]—capturing functional-correctness properties by simultaneously reasoning about multiple system executions. For example, consider a credit-screening model of a bank. The expected property that "if a person is denied a loan and their income decreases, they should still be denied the loan", or conversely "if a person is granted a loan and their income increases, they should still be granted the loan", is a 2-safety hyperproperty—we need two model executions to validate its correctness. In contrast, the property that "a person with no income should be denied a loan" is a standard (1-)safety property since it can be validated by individual model executions. Overall, k-safety hyperproperties generalize standard safety properties in that they require reasoning about k different executions.

Examples. Although we are not the first to observe that hyperproperties can be used to specify ML models (e.g., [Seshia et al., 2018; Sharma and Wehrheim, 2020]), we go a step further by demonstrating the wide applicability of general, user-provided k-safety properties for such models. We use examples from five distinct domains throughout this paper:

Tabular data. Consider the COMPAS dataset [Larson et al., 2016], which determines how likely criminals are to re-offend. An expected hyperproperty for models trained on COMPAS could be that "if the number of committed felonies for a given criminal increases, then their recidivism risk should not decrease". Note that this is essentially monotonicity in an input feature, a special case of the hyperproperties we consider here.

Images. Using the MNIST dataset [LeCun et al., 1999], which classifies images of handwritten digits, an expected
hyperproperty could be that "if a blurred image is correctly classified, then its unblurred version should also be correctly classified". Note that this is not monotonicity in a feature as whether or not an image is blurred does not constitute part of the model input (i.e., the image); instead, blurring may affect most, if not all, pixels.

**Speech.** Similarly, for the SpeechCommand dataset [Warden, 2018], which classifies short spoken commands, an expected hyperproperty could be that "if a speech command with white noise is correctly classified, then its non-noisy version should also be correctly classified".

**Natural language.** The HotelReview dataset [Liu, 2017] is used for sentiment analysis of hotel reviews. An expected hyperproperty could be that "if a review becomes more negative, the sentiment should not become more positive". Note that, again, making a review more negative may significantly affect the model input.

**Action policies.** LunarLander is a popular Gym [Brockman et al., 2016] environment consisting of a 2D-world with an uneven lunar surface and a reinforcement-learning (RL) lander, which initially appears far above the surface and moves downward. The goal is to navigate and land the lander on its two legs; if the body ever touches the surface, the lander crashes. An expected hyperproperty could be that "if the lander lands successfully, then decreasing the surface height (thus giving the lander more time to land) should also result in landing successfully". Here, even a seemingly simple change to the initial game state may result in significant changes to subsequent states since the policy is invoked repeatedly during the game.

In practice, such properties are defined by users, thus expressing model expectations that are deemed important in their particular usage scenario.

**Approach and contributions.** In this paper, we show the wide applicability of $k$-safety properties for ML models. We design a declarative, domain-agnostic specification language, NOMOS ("law" in Greek), for writing them. In contrast to existing approaches, NOMOS can express general $k$-safety properties capturing arbitrary relations between more than one input-output pair; these subsume the more specific relations of robustness, fairness, and monotonicity.

We further design a fully automated framework (see Fig. 3) for validating NOMOS properties using metamorphic testing [Chen et al., 1998; Segura et al., 2016]. On a high-level, our framework takes as input the model under test and a set of $k$-safety properties for the model. As output, it produces tests for which the model violates the specified properties. Note that a single test for a $k$-safety property consists of $k$ concrete inputs to the model under test. Under the hood, the translator component of the framework compiles the provided NOMOS properties into a test harness, i.e., software that tests the given model against the properties. The harness employs a test generator for generating inputs to the model using metamorphic testing and an oracle for detecting property violations.

In summary, this paper makes the following contributions:

- We present NOMOS, the first specification language for expressing general $k$-safety hyperproperties for ML models, naturally opening up the possibility to apply various validation or verification techniques for checking such properties.
- We demonstrate the wide applicability of such properties through case studies from several domains and the expressiveness of our language in capturing them.
- We design and implement a fully automated, publicly available framework\(^1\) for validating such properties using metamorphic testing.
- We evaluate the effectiveness of our testing framework in detecting property violations across a broad range of different domains. We also perform a feasibility study to showcase how such violations can be used to improve model training.

### 2 NOMOS Specification Language

NOMOS allows a user to specify $k$-safety properties over source code invoking an ML model under test. On a high level, a NOMOS specification consists of three parts: (1) the precondition, (2) the source code—Python in our implementation—invoking the model, and (3) the postcondition. Pre- and postconditions are commonly used in formal methods, for instance, in Hoare logic [Hoare, 1969] and design by contract [Meyer, 1992]. Here, we adapt pre- and postconditions for reasoning about $k$-safety properties of ML models.

The precondition captures the conditions under which the model should be invoked, allowing the user to express arbitrary relations between more than one model input. It is expressed using zero or more requires statements, each capturing a condition over inputs; the logical conjunction of these conditions constitutes the precondition. The source code may be arbitrary code invoking the model one or more times to capture $k$ input-output pairs. Finally, the postcondition captures the safety property that the model is expected to satisfy. It is expressed using zero or more ensures statements, each taking a condition that, unlike for the precondition, may refer to model outputs; the logical conjunction of these conditions constitutes the postcondition.

**Examples.** Consider the NOMOS specification of Fig. 1a expressing the COMPAS property described earlier. On line 1, we specify that we need an input $x_1$, i.e., a criminal. Lines 2–4 get the first feature of $x_1$, which corresponds to the number of felonies, and assign it to variable $v_1$; in variable $v_2$, we increase this number, and create a new criminal $x_2$ that differs from $x_1$ only with respect to this feature, i.e., $x_2$ has committed more felonies than $x_1$. Line 5 specifies a precondition that the new criminal’s felonies should not exceed a sensible limit. Lines 6–7 declare two outputs, $d_1$ and $d_2$, that are assigned the model’s prediction when calling it with criminal $x_1$ and $x_2$, respectively (see block of Python code on lines 8–11). Finally, on line 13, we specify the postcondition that the recidivism risk of criminal $x_2$ should not be lower than that of $x_1$.

Fig. 1b shows the MNIST specification. Given an image $x_1$ (line 1), image $x_2$ is its blurred version (line 2), and variable $v_1$ contains its correct label (line 3), e.g., retrieved from the dataset. Note that functions such as blur and label extend the core NOMOS language and may be easily added by the user.

\(^1\)https://github.com/Rigorous-Software-Engineering/nomos
ensures $d_2 = d_1 = 10$

# 0-pos, 1-neg
$d_3 = \{ d_1 \leq d_3 \} \Rightarrow (x_1, 1, v_2)$
# output $d_1$;
# output $d_2$;
8 {
9 $d_1 = \text{predict}(x_1)$
10 $d_2 = \text{predict}(x_2)$
11 $
12 \# 0-low, 1-medium, 2-high risk
13 \text{ensures } d_1 \leq d_2;

(a) COMPAS 2-safety property.

1 input $x_1$;
2 input $x_2$;
3 var $v_1 := \text{getFeat}(x_1, 1)$;
4 var $v_2 := \text{getFeat}(x_2, 1)$;
5 var $v_3 := \text{strConcat}([v_1, v_2])$;
6 output $d_1$;
7 output $d_3$;
8 {
9 $d_1 = \text{predict}(x_1)$
10 $d_3 = \text{predict}(x_3)$
11 }$
12 \# 0-pos, 1-neg
13 \text{ensures } d_1 \leq d_3;

(c) HotelReview 2-safety property.

Figure 1: Example $k$-safety specifications in NOMOS.

The postcondition on line 10 says that if the blurred image is correctly classified, then so should the original image. Note that we defined a very similar specification for the SpeechCommand property—instead of $\text{blur}$, we used function $\text{wNoise}$ adding white noise to audio.

The HotelReview specification is shown in Fig. 1c. A hotel review consists of a positive and a negative section, where a guest describes what they liked and did not like, respectively. On lines 1–2, we obtain two reviews, $x_1$ and $x_2$, and in variables $v_1$ and $v_2$ on lines 3–4, we store their negative sections (feature 1 retrieved with function $\text{getFeat}$). We then create a third variable, $x_3$, which is the same as $x_1$ except that its negative section is the concatenation of $v_1$ and $v_2$ (lines 5–6). The postcondition on line 14 checks that the detected sentiment is not more positive for review $x_3$ than for $x_1$.

Finally, consider the LunarLander specification in Fig. 1d. On line 1, we obtain an input $s_1$, which is an initial state of the game. Line 2 "relaxes" this state to obtain a new state $s_2$, which differs from $s_1$ only in that the height of the lunar surface is lower. In the block of Python code that follows (lines 5–11), we initialize outputs $o_1$ and $o_2$ to zero and play the game from each initial state, $s_1$ and $s_2$, in a loop; $o_1$ and $o_2$ accumulate the number of wins. We use a loop because the environment is stochastic—firing an engine of the lander follows a probability distribution. Therefore, by changing the environment random seed $rs$ on line 8, we take stochasticity into account. In each loop iteration however, we ensure that the game starting from $s_2$ is indeed easier, i.e., that stochasticity cannot make it harder, by using the same seed on lines 9–10. Note that function $\text{play}$ invokes the policy multiple times (i.e., after every step in the game simulator). Finally, line 13 ensures that, when playing the easier game (starting with $s_2$), the number of wins should not decrease. Since this property depends on 20 model invocations, it is a 20-safety property! Conversely, we can also make the game harder by "unrelaxing" the original initial state, i.e., increasing the surface height.

Grammar. Fig. 2 provides a formal grammar for NOMOS (in a variant of extended Backus-Naur form). The top-level construct is $\langle\text{spec}\rangle$ on lines 1–4. It consists of zero or more $\langle\text{import}\rangle$ statements—the curly braces denote repetition—to import source-code files containing custom implementations for domain-specific functions, e.g., $\text{blur}$ or $\text{wNoise}$, one or more input declarations, variable declarations, preconditions, output declarations, the source-code block, and postconditions. We define these sub-constructs in subsequent rules (lines 6–11). For instance, a precondition (line 9 of Fig. 2) consists of the token $\langle\text{requires}\rangle$, a Boolean expression, and a semicolon. For brevity, we omit a definition of $\langle\text{code}\rangle$; it denotes arbitrary Python code that is intended to invoke the model under test and assign values to output variables. We additionally omit the basic identifiers $\langle\text{model_name}\rangle$ and $\langle\text{var_name}\rangle$.

The grammar also defines various types of expressions needed in the above sub-constructs. In their definitions, we use the $|$ combinator to denote alternatives. In particular, we define scalar (lines 12–18), Boolean (lines 19–25), and record expressions (lines 26–33). The latter express complex object-like values, e.g., images or game states. In the definitions, we include extensions to the core language with domain-specific functions supporting the application domains considered here—e.g., $\text{getFeat}$ and $\text{setFeat}$ retrieve and modify record fields. Integer and string expressions are defined as expected.
Metamorphic testing [Chen et al., 1998; Segura et al., 2016] is a testing technique that addresses the lack of an existing oracle defining correct system behavior. Specifically, given an input, metamorphic testing transforms it such that the relation between the outputs (i.e., the output of the system under test when executed on the original input and the corresponding output when executed on the transformed input) is known. If this relation between outputs does not hold, then a bug is detected. As a simple example, consider testing a database system; given a query as the original input, assume that the transformed input is the same query with weakened constraints. A bug is detected if the transformed query returns fewer results than the original one, which is more restrictive. So far, metamorphic testing has been used to test ML models from specific application domains, e.g., image classifiers [Dwarakanath et al., 2018; Tian et al., 2020], translation systems [via Isotopic Replacement, 2022], NLP models [Ma et al., 2020], object-detection systems [Wang and Su, 2020], action policies [Eniser et al., 2022], and autonomous cars [Tian et al., 2018; Zhang et al., 2018].

In our setting, we observe that metamorphic testing is a natural choice for validating general k-safety properties as these also prescribe input transformations and expected output relations. For instance, in Fig. 1a, lines 2–5 describe the transformation to input x1 in order to obtain x2, and line 13 specifies the relation between the corresponding outputs. We, therefore, design the framework in Fig. 3 for validating a model against a NOMOS specification using metamorphic testing. The output of our framework is a set of (unique) bugs, i.e., test cases revealing postcondition violations. For Fig. 1a, a bug would comprise two concrete instances of a criminal, c1 and c2, such that (1) c2 differs from c1 only in having more felonies, and (2) the recidivism risk of c2 is predicted to be lower than that of c1.

Under the hood, the translator component of the framework compiles the NOMOS specification into a test harness, i.e., a Python program that tests the model against the specified properties. Our implementation parses NOMOS specifications using an ANTLR4 [Parr, 2013] grammar. After semantically checking the parsed abstract syntax tree (AST), our framework translates the AST into the Python program constituting the test harness. A snippet of the generated harness for the specification of Fig. 1a is shown in Fig. 4. The test harness employs a test generator and an oracle component, for generating inputs to the model using metamorphic testing and for detecting postcondition violations, respectively.

As shown in Fig. 4, the model is tested until a user-specified budget is depleted (line 1). In each iteration of this loop, the test generator creates k model inputs that satisfy the given precondition, if any (lines 3-11). Specifically, for every input declaration, the test generator randomly selects an input from a source specified in the imported files (line 4)—note that import statements are not shown here but are defined on line 5 of Fig. 2. In our evaluation, we have used both the test set and the output of an off-the-shelf fuzzer [Eniser et al., 2022] as such input sources. The metamorphic transformation of an input can be performed through var declarations, which are compiled into temporary variables in the harness (lines 5–7). Before the test generator returns the k generated model inputs, the specified precondition is checked; if it is violated, the process repeats until it holds (lines 9–11).

Next, the block of Python code in the specification is executed (lines 12–14), and finally the oracle component checks the postcondition (lines 16–21). On line 21, the oracle records each detected bug and processes it for subsequent duplication. In particular, for each bug, the oracle hashes any source of randomness in generating the model inputs (i.e., for the example of Fig. 4, there is randomness on lines 4 and 6).
Two bugs are considered duplicate if their hashes match, that is, if the generated model inputs are equivalent. Note that we avoid comparing model inputs directly due to their potential complexity, \textit{e.g.}, in the case of game states.

Our framework allows users to express specifications much more concisely than if they were writing the test harnesses themselves; for instance, for our case studies in several domains, the test harnesses are between 5.2x and 6.3x larger (counting non-whitespace characters) than the corresponding NOMOS specifications.

4 Experimental Evaluation

So far, we have demonstrated the expressiveness of NOMOS by specifying hyperproperties for models in diverse domains. This section focuses on evaluating the effectiveness of our testing framework in finding bugs. We describe the benchmarks, experimental setup, and results. We also present a feasibility study on how detected bugs can improve model training.

Benchmarks. We trained models using seven datasets from five application domains as follows:

\textbf{Tabular data.} We used the COMPAS [Larson et al., 2016] and GermanCredit [Hofmann, 1994] datasets; the latter classifies people based on their credit risk. For the COMPAS dataset, we trained a fully connected neural network (NN) with 3 hidden layers of size 12, 9, and 9 and ca. 1100 parameters as well as a decision tree (DT) with \textit{max\_depth} = 8. For GermanCredit, we trained an NN with 1 hidden layer of size 10 and ca. 1400 parameters as well as a DT with \textit{max\_depth} = 6. For COM-PAS, we achieved 74\% (NN) and 72\% (DT) accuracy, and for GermanCredit, 78\% (NN) and 70\% (DT). Note that, even though we report accuracy here, the achieved score does not necessarily affect whether a specified property holds, \textit{i.e.}, a perfectly accurate model could violate the property, whereas a less accurate model might not.

\textbf{Images.} Using the MNIST dataset [LeCun et al., 1999], we trained a convolutional NN with LeNet-5 architecture [LeCun et al., 1998] achieving 98\% accuracy.

\textbf{Speech.} We pre-processed the SpeechCommand dataset [Warden, 2018] to convert waveforms to spectrograms, showing frequency changes over time. As these are typically represented as 2D-images, we trained a convolutional NN classifying spectrogram images; it consists of 2 convolutional layers with kernels (32x32x3) and (64x64x3) as well as a fully connected layer of size 128, and has ca. 1.6M parameters. The model achieves 84\% test accuracy.

\textbf{Natural language.} For the HotelReview dataset [Liu, 2017], we used a pre-trained Universal Sentence Encoder (USE) [Cer et al., 2018] to encode natural-language text into high dimensional vectors. USE compresses any textual data into a vector of size 512 while preserving the similarity between sentences. We trained a fully connected NN with 2 hidden layers (256 and 128 neurons, respectively), ca. 160K parameters, and an accuracy of 82\% on the encoded hotel reviews.

\textbf{Action policies.} In LunarLander [Brockman et al., 2016], touching a leg of the lander to the surface yields reward +100, whereas touching the body yields −100; the best-case reward is over 200. We trained an RL policy that achieves an average reward of ca. 200. We also used BipedalWalker, another popular Gym [Brockman et al., 2016] environment where the aim is to train a bipedal robot to walk until the end of a rough terrain. Moving forward yields positive reward, totaling over 300 at the end of the terrain; falling yields −100. We trained an RL policy that achieves an average reward of ca. 300.

\textbf{Experimental setup.} For each of these models, we wrote one or more NOMOS specifications to capture potentially desired properties, for a total of 32 properties (see Appx. A for a complete list).

Each test harness used a budget of 5000 (see line 1 of Fig. 4), that is, it generated 5000 test cases satisfying the precondition, if any. We ran each harness with 10 different random seeds to account for randomness in the testing procedure. Here, we report arithmetic means (\textit{e.g.}, for the number of bugs) unless stated otherwise. In all harnesses except for LunarLander and BipedalWalker, the input source (\textit{e.g.}, line 4 of Fig. 4) is the test set. For LunarLander and BipedalWalker, the input source is a pool of game states that was generated by \textit{π}-fuzz [Eniser et al., 2022] after fuzzing our policy for 2 hours.

We used a machine with a Quadro RTX 8000 GPU and an Intel(R) Xeon(R) Gold 6248R CPU @ 3.00GHz for training models and running tests. Running 5000 tests takes a few seconds for DTs. It takes longer for NNs (excluding action policies), ranging from 5 to 20 minutes, depending on the specification and dataset; note, however, that most of this time is spent on querying the models, \textit{i.e.}, between 93\% and 99\% of the testing time. For action policies, testing takes significantly longer as the NN is called multiple times during a single test. Specifically, it takes up to 2.5 hours for LunarLander and up
Table 1: Number of specified properties, violated properties, and unique bugs per dataset and model.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Model</th>
<th>Properties Specified</th>
<th>Violated</th>
<th>Unique Bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPAS</td>
<td>NN</td>
<td>12</td>
<td>7</td>
<td>960.0</td>
</tr>
<tr>
<td></td>
<td>DT</td>
<td>12</td>
<td>6</td>
<td>294.8</td>
</tr>
<tr>
<td>GermanCredit</td>
<td>NN</td>
<td>10</td>
<td>6</td>
<td>295.2</td>
</tr>
<tr>
<td></td>
<td>DT</td>
<td>10</td>
<td>6</td>
<td>286.9</td>
</tr>
<tr>
<td>MNIST</td>
<td>NN</td>
<td>1</td>
<td>1</td>
<td>15.6</td>
</tr>
<tr>
<td>SpechCommand</td>
<td>NN</td>
<td>1</td>
<td>1</td>
<td>14.2</td>
</tr>
<tr>
<td>HotelReview</td>
<td>NN</td>
<td>4</td>
<td>4</td>
<td>3288.0</td>
</tr>
<tr>
<td>LunarLander</td>
<td>NN</td>
<td>2</td>
<td>2</td>
<td>3459.0</td>
</tr>
<tr>
<td>BipedalWalker</td>
<td>NN</td>
<td>2</td>
<td>2</td>
<td>443.3</td>
</tr>
</tbody>
</table>

Table 2: Average number of unique bugs for each specification.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Specification</th>
<th>Unique Bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPAS</td>
<td>Felony Inc</td>
<td>42.9</td>
</tr>
<tr>
<td></td>
<td>Felony Dec</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>Misdmr Inc</td>
<td>619.7</td>
</tr>
<tr>
<td></td>
<td>Misdmr Dec</td>
<td>4.0</td>
</tr>
<tr>
<td></td>
<td>Priors Inc</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>Priors Dec</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>Others Inc</td>
<td>289.0</td>
</tr>
<tr>
<td></td>
<td>Others Dec</td>
<td>3.0</td>
</tr>
<tr>
<td></td>
<td>IsRecid Set</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>IsRecid Unset</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>IsVRecid Set</td>
<td>0.9</td>
</tr>
<tr>
<td></td>
<td>IsVRecid Unset</td>
<td>0.0</td>
</tr>
<tr>
<td>GermanCredit</td>
<td>Crdt Amount Inc</td>
<td>122.3</td>
</tr>
<tr>
<td></td>
<td>Crdt Hist Inc</td>
<td>78.1</td>
</tr>
<tr>
<td></td>
<td>Crdt Hist Dec</td>
<td>122.8</td>
</tr>
<tr>
<td></td>
<td>Empl Since Inc</td>
<td>13.5</td>
</tr>
<tr>
<td></td>
<td>Empl Since Dec</td>
<td>30.9</td>
</tr>
<tr>
<td></td>
<td>Install Rate Inc</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>Job Inc</td>
<td>47.9</td>
</tr>
<tr>
<td></td>
<td>Job Dec</td>
<td>2.0</td>
</tr>
<tr>
<td>MNIST</td>
<td>Blur</td>
<td>15.6</td>
</tr>
<tr>
<td>SpeechCommand</td>
<td>WNoise</td>
<td>14.2</td>
</tr>
<tr>
<td>HotelReview</td>
<td>Post-Del</td>
<td>861.1</td>
</tr>
<tr>
<td></td>
<td>Post-Add</td>
<td>876.1</td>
</tr>
<tr>
<td></td>
<td>Neg-Del</td>
<td>756.2</td>
</tr>
<tr>
<td></td>
<td>Neg-Add</td>
<td>794.6</td>
</tr>
<tr>
<td>LunarLander</td>
<td>Relax</td>
<td>124.3</td>
</tr>
<tr>
<td></td>
<td>Unrelax</td>
<td>3334.5</td>
</tr>
<tr>
<td>BipedalWalker</td>
<td>Relax</td>
<td>290.5</td>
</tr>
<tr>
<td></td>
<td>Unrelax</td>
<td>153.0</td>
</tr>
</tbody>
</table>

Table 3: Minimum-bug and maximum-reward policies generated with normal and guided training.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Minimum-Bug Policy</th>
<th>Maximum-Reward Policy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bugs</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Results. Tab. 1 gives an overview of the number of specified properties, violated properties, and unique bugs per dataset and model. Our framework was able to find violations for all datasets, and in particular, for 26 of these properties—see Tab. 2. Most violations were exhibited through tens or hundreds of unique tests. This demonstrates that our framework is effective in detecting bugs even with as few as 5000 tests per property; in contrast, fuzzers for software systems often generate millions of tests before uncovering a bug.

The average number of bugs per property varies significantly depending on the property, model, and dataset (see Tab. 2). For instance, for COMPAS, the average number of bugs ranges from 0.5 to 619.7 when testing the NN classifier against each of the twelve different properties.

There are six properties that were not violated by any model trained on COMPAS and GermanCredit. For four of these, we observed that the involved features almost never affect the outcome of our models, thereby trivially satisfying the properties. In the other cases, the training data seems to be sufficient in ensuring that the properties hold for the models.

Feasibility study. Our results show that our framework is effective in detecting property violations. But are these violations actionable? A natural next step is to use them for repairing the model under test or incorporate them when training the model from scratch—much like adversarial neural network training for robustness issues [Madry et al., 2018]. While a comprehensive exploration of such options is beyond the scope of this work, we performed a feasibility study to investigate whether the reported violations are actionable.

For this study, we selected LunarLander due to its higher complexity. On a high level, we incorporated buggy game states, i.e., ones that resulted in property violations, during policy training. In particular, we adjusted the existing training algorithm [PO [Schulman et al., 2017]] to start episodes not only from random initial states, but also from buggy states. As training progresses, our guided-training algorithm gradually increases the probability of starting from buggy states. The intuition behind this choice is to focus more on “ironing out” bugs toward the end of the training, when the policy is already able to achieve decent rewards.

Under the hood, our guided-training algorithm tests the current policy at regular intervals (every 5 rollouts in our experiments), essentially alternating between training and testing phases. Any bugs found during the latest testing phase are added to a pool of buggy states from which the algorithm selects initial states during subsequent training phases. We prioritize most recently detected buggy states, but we also include older bugs to ensure the policy does not "forget".

For our experiments, we trained 10 policies with each training algorithm, i.e., normal and guided. Tab. 3 summarizes the policies that were generated during these training runs—each row corresponds to a training run, and for each run, we save and test the current policy every 10K time steps. In the four leftmost columns, we focus on policies with the fewest number of bugs. The first two columns show the number of bugs and...
with the same reward, we show the one with fewer bugs. It was later observed that hyperproperties can also be useful not only for assessing the quality of a model, but also for training better models. Wehrheim et al. [Sharma and Wehrheim, 2020], introduce verification-based testing of monotonicity in ML models. A framework in detecting property violations. Although users from several domains and evaluated the effectiveness of our techniques for checking such properties. We have demonstrated the possibility to apply other validation or verification techniques for checking such properties. We have demonstrated up the possibility to apply other validation or verification techniques for checking such properties. We have demonstrated the wide applicability of such properties through case studies such as robustness and fairness. It, therefore, naturally opens up the possibility to apply other validation or verification techniques for checking such properties. We have demonstrated the wide applicability of such properties through case studies from several domains and evaluated the effectiveness of our framework in detecting property violations. Although users could manually write test cases or a test harness for each desired property, this would be tedious, repetitive, and easy to

to specify ML models (e.g., [Seshia et al., 2018; Sharma and Wehrheim, 2020]). However, no prior work has explored how to specify general, user-provided $k$-safety properties for ML models and how to leverage these specifications for automated testing. In the following, we give an overview of existing verification and testing techniques for ML models.

**Verification.** Numerous techniques verify specific functional-correctness properties of models, such as robustness (e.g., [Huang et al., 2017; Gehr et al., 2018; Singh et al., 2019; Berrada et al., 2021; Wang et al., 2021; Yang et al., 2021; Li et al., 2020]), fairness (e.g., [Albarghouthi et al., 2017; Bastani et al., 2019; Urban et al., 2020]), and others (e.g., [Katz et al., 2017; Wang et al., 2018]). Here, we do not target verification of $k$-safety properties, however in principle, NOMOS specifications could be used to capture proof obligations for verifiers.

**Testing.** Testing ML models is extensively studied, including techniques for testing fairness (e.g., [Udeshi et al., 2018; Ma et al., 2020; Zhang et al., 2020]) and robustness (e.g., [Wicker et al., 2018; Sun et al., 2018; Usman et al., 2021]). There is also work using metamorphic testing to find robustness issues in specific domains, such as autonomous driving [Tian et al., 2018; Zhang et al., 2018], object detection [Zhou and Sun, 2019], and translation [He et al., 2020].

Beyond robustness and fairness, Sharma and Wehrheim [Sharma and Wehrheim, 2020], introduce verification-based testing of monotonicity in ML models. A model is said to be monotone with respect to an input feature if an increase in the feature implies an increase in the model’s prediction, e.g., the higher the income, the larger the loan. Deng et. al. also focus on specifying and testing monotonicity properties in autonomous driving [Deng et al., 2021; Deng et al., 2022]. Although certain popular robustness, fairness, and monotonicity properties do constitute 2-safety properties (e.g., slightly perturbing the pixels of an image should not change its classification, or changing the race of a criminal should not make them more or less likely to re-offend), none of this work targets general hyperproperties.

In this paper, we use metamorphic testing to effectively and efficiently find bugs in ML models, but the specific testing technique is not the main contribution of our work. We designed our framework to be modular such that its test generator component may be instantiated with other techniques.

6 Conclusion and Outlook

We have presented the NOMOS language for specifying $k$-safety properties of ML models and an automated testing framework for detecting violations of such properties. NOMOS is the first high-level specification language for expressing general hyperproperties of models, subsuming more specific ones such as robustness and fairness. It, therefore, naturally opens up the possibility to apply other validation or verification techniques for checking such properties. We have demonstrated the wide applicability of such properties through case studies from several domains and evaluated the effectiveness of our framework in detecting property violations.
get wrong; it would also be difficult to update and extend properties if needed. In contrast, our NOMOS specifications are concise and enable users to think about properties on a higher level of abstraction.

There are several promising directions for future work. For the ML community, model repair and guided training might be the most interesting direction for building on NOMOS and our testing framework. One way to think about specifications is as a, possibly infinite, source of training examples. Our feasibility study has already provided some empirical evidence for how such examples can be incorporated in the training process. However, more work is needed, and adversarial-training techniques could be adapted to improve the effectiveness.

For the testing community, an interesting direction could be to explore more effective input-generation techniques, such as coverage-guided testing. This may reduce the testing time or increase the number of bugs that can be found within a given time budget. Such advances can be crucial for reducing the testing overhead when performing guided training.

For the formal-methods community, a natural next step is to build verification tools for certifying that a property holds for all inputs. This could be particularly promising for models used in safety-critical domains, such as autonomous driving.

We believe that NOMOS can bring these communities together to facilitate developing functionally correct models.

### A Specifications

**COMPAS**

- **COMPASFelony Inc.** If the number of committed felonies for a criminal increases, then their recidivism risk should not decrease.
- **COMPASFelony Dec.** If the number of committed felonies for a criminal decreases, then their recidivism risk should not increase.
- **COMPASMisdmr Inc.** If the number of committed misdemeanors for a criminal increases, then their recidivism risk should not decrease.
- **COMPASMisdmr Dec.** If the number of committed misdemeanors for a criminal decreases, then their recidivism risk should not increase.
- **COMPASPrriors Inc.** If the number of priors for a criminal increases, then their recidivism risk should not decrease.
- **COMPASPrriors Dec.** If the number of priors for a criminal decreases, then their recidivism risk should not increase.
- **COMPASOthers Inc.** If the number of other crimes committed by a criminal increases, then their recidivism risk should not decrease.
- **COMPASOthers Dec.** If the number of other crimes committed by a criminal decreases, then their recidivism risk should not increase.
- **COMPASlsRecid Set.** If a criminal becomes a recidivist, then their recidivism risk should not decrease.
- **COMPASlsRecid Unset.** If a criminal ceases to be a recidivist, then their recidivism risk should not increase.
- **COMPASlsVRecid Set.** If a criminal becomes a violent recidivist, then their recidivism risk should not decrease.
- **COMPASlsVRecid Unset.** If a criminal ceases to be a violent recidivist, then their recidivism risk should not increase.

**GermanCredit**

- **EInstall Rate Inc.** If the credit amount requested by a person increases, then they should not be more likely to receive it.
- **EInstall Rate Dec.** If the credit amount requested by a person decreases, then they should not be less likely to receive it.
- **Hist Inc.** If a person’s credit history worsens, then they should not be more likely to receive credit.
- **Hist Dec.** If a person’s credit history improves, then they should not be less likely to receive credit.
- **Empl Since Inc.** If a person’s employment years increase, then they should not be less likely to receive credit.
- **Empl Since Dec.** If a person’s employment years decrease, then they should not be more likely to receive credit.
- **Job Inc.** If a person is promoted, then they should not be less likely to receive credit.
- **Job Dec.** If a person is demoted, then they should not be more likely to receive credit.
- **Blur.** If a blurred image is correctly classified, then its unblurred version should also be correctly classified.
- **WNoise.** If a speech command with white noise is correctly classified, then its non-noisy version should also be correctly classified.
- **Pos-Del.** Deleting the positive comments of a hotel review should not make it more positive.
- **Pos-Add.** If more positive comments are added to a hotel review, it should not become more negative.
- **Neg-Del.** Deleting the negative comments of a hotel review should not make it more negative.
- **Neg-Add.** If more negative comments are added to a hotel review, it should not become more positive.
- **Relax.** If the lander lands successfully, then increasing the surface height (thus giving the lander more time to land) should also result in landing successfully.
- **Unrelax.** If the lander fails to land, then decreasing the surface height (thus giving the lander less time to land) should also result in failing to land.
- **BipedalWalkerRelax.** If the walker successfully reaches the end of the terrain, then making the terrain smoother should also result in successfully reaching the end.
- **BipedalWalkerUnrelax.** If the walker fails to reach the end of the terrain, then making the terrain rougher should also result in failing to reach the end.

### Ethical Statement

There are no ethical issues.

### Acknowledgments

This work was supported by DFG grant 389792660 as part of TRR 248 (see https://perspicuous-computing.science) and Maria Christakis’ Google Research Scholar Award 2022.
References


