Specifying and Fuzzing Machine-Learning Models

Thesis approved by
the Department of Computer Science
University of Kaiserslautern-Landau
for the award of the Doctoral Degree
Doctor of Engineering (Dr.-Ing.)

to

Hasan Ferit Enişer

Date of Defense:	13 / 06 / 2025
Dean:	Prof. Dr. Christoph Garth
Reviewers:	Prof. Dr. Maria Christakis
	Prof. Dr. Rupak Majumdar
	Prof. Dr. Joerg Hoffmann

DE-386

To my children...

Abstract

Machine Learning (ML) models are increasingly prevalent in safety-critical systems, from self-driving cars to aviation, where failures can result in catastrophic outcomes. While researchers have addressed specific properties like robustness and fairness, specifying and checking general functional-correctness expectations from ML models remains challenging. This thesis introduces novel tools and approaches inspired by software testing concepts to specify and fuzz ML artifacts for their functional correctness. Software testing identifies bugs by running programs with given inputs, facing two main challenges: generating test inputs and finding test oracles. Fuzzing is a widely adopted method for generating test inputs, while specifications address the oracle problem. These techniques and concepts have proven effective and crucial for validating software reliability. We tailor these methods to assess ML model reliability. One of the biggest recent advancements in machine learning has been in solving sequential decision-making problems where agents learn action policies. In this thesis, we devise techniques to test action policies' reliability. Beyond checking if policies lead to undesirable outcomes, we address: *how can we identify undesirable yet avoidable outcomes?* We present novel test oracles based on metamorphic relations and develop the $\pi-$fuzz framework to identify bugs in action policies. Next, we formalize metamorphic relations as k-safety properties, or *hyper*properties, describing relationships between multiple input-output pairs. We show hyperproperties can specify functional correctness across various ML domains. To express these, we create NOMOS, a declarative, domain-agnostic specification language with an automated testing framework. We demonstrate its effectiveness in finding bugs across various domains including image classification, sentiment analysis, and speech recognition. We also extend NOMOS to accommodate code translation models. Overall, this thesis contributes to the field by providing a specification language and novel automated testing frameworks to validate the reliability and safety of ML models which are now prevalent in our daily lives.

Kurzfassung

Machine Learning (ML) Modelle sind zunehmend in sicherheitskritischen Systemen verbreitet, von selbstfahrenden Autos bis zur Luftfahrt, wo Ausfälle zu katastrophalen Folgen führen können. Während Forscher spezifische Eigenschaften wie Robustheit und Fairness behandelt haben, bleibt die Spezifikation und Überprüfung allgemeiner funktionaler Korrektheitserwartungen von ML-Modellen eine Herausforderung. Diese Arbeit stellt neuartige Werkzeuge und Ansätze vor, die von Konzepten des Software-Testens inspiriert sind, um ML-Artefakte bezüglich ihrer funktionalen Korrektheit zu spezifizieren und zu fuzzen. Software-Testing identifiziert Fehler durch das Ausführen von Programmen mit gegebenen Eingaben und steht vor zwei Hauptherausforderungen: der Generierung von Testeingaben und dem Finden von Test-Orakeln. Fuzzing ist eine weit verbreitete Methode zur Generierung von Testeingaben, während Spezifikationen das Orakel-Problem adressieren. Diese Techniken und Konzepte haben sich als effektiv und entscheidend für die Validierung der Software-Zuverlässigkeit erwiesen. Wir passen diese Methoden an, um die Zuverlässigkeit von ML-Modellen zu bewerten. Eine der größten jüngsten Fortschritte im maschinellen Lernen war die Lösung sequenzieller Entscheidungsprobleme, bei denen Agenten Handlungsrichtlinien lernen. In dieser Arbeit entwickeln wir Techniken zur Prüfung der Zuverlässigkeit von Handlungsrichtlinien. Über die Überprüfung hinaus, ob Richtlinien zu unerwünschten Ergebnissen führen, behandeln wir die Frage: *wie können wir unerwünschte aber vermeidbare Ergebnisse identifizieren?* Wir präsentieren neuartige Test-Orakel basierend auf metamorphen Beziehungen und entwickeln das $\pi-$fuzz Framework zur Identifikation von Fehlern in Handlungsrichtlinien. Als nächstes formalisieren wir metamorphe Beziehungen als *k*-Sicherheitseigenschaften oder *Hyper*eigenschaften, die Beziehungen zwischen mehreren Eingabe-Ausgabe-Paaren beschreiben. Wir zeigen, dass Hypereigenschaften die funktionale Korrektheit in verschiedenen ML-Domänen spezifizieren können. Um diese auszudrücken, erstellen wir NOMOS, eine deklarative, domänenagnostische Spezifikationssprache mit einem automatisierten Test-Framework. Wir demonstrieren ihre Wirksamkeit beim Finden von Fehlern in verschiedenen Domänen einschließlich Bildklassifikation, Sentimentanalyse und Spracherkennung. Wir erweitern NOMOS auch für Code-Übersetzungsmodelle. Insgesamt trägt diese Arbeit zum Fachgebiet bei, indem sie eine Spezifikationssprache und neuartige automatisierte Test-Frameworks zur Validierung der Zuverlässigkeit und Sicherheit von ML-Modellen bereitstellt, die heute in unserem täglichen Leben weit verbreitet sind.

Acknowledgments

This thesis represents one of the greatest achievements of my life. The academic journey to reach this point has been incredibly challenging, yet it has granted me invaluable experiences and skills along the way. I would like to express my heartfelt gratitude to everyone who made it possible. Though words fall short, I will try my best to convey my thanks. If I have unintentionally omitted anyone, I sincerely apologize.

First and foremost, I owe deep gratitude to my advisor, Maria Christakis. Her constant support, approachability, and kindness were crucial throughout my journey. In moments of doubt or failure, her reassurances gave me the strength to keep moving forward, and her belief in my potential was a guiding light. She helped me persevere, even though a significant part of my PhD had to be conducted remotely. I look forward to continuing our collaboration in the next chapter.

A special thank you to our collaborator, Valentin Wüstholz, whose rigorous and thought-provoking discussions challenged me to think critically and grow as a researcher, and for that, I hold him in the highest regard. I was also fortunate to collaborate with Jörg Hoffmann and Adish Singla, whose insights and expertise enriched my work in countless ways.

I extend my heartfelt thanks to my thesis reviewers, Maria Christakis, Rupak Majumdar and Jörg Hoffmann, for their time and thoughtful feedback. I am also grateful to my PhD committee members, Maria Christakis, Rupak Majumdar and Jörg Hoffmann, and the chairperson Annette Bieniusa for ensuring the thesis evaluation process was conducted smoothly.

A big thank you to the incredible administrative staff at MPI-SWS in Kaiserslautern: Corinna, Ruth, Geraldine, Susanne, Vera and Mary-Lou. Their efficiency and dedication ensured that non-academic matters were seamless and stress-free. I am equally grateful to Tobias Kaufmann, whose expertise in IT made everything run effortlessly.

The five years I spent in Kaiserslautern were some of the most memorable years of my life, thanks to the amazing friends I made along the way. They made my journey unforgettable; together, we created cherished memories through travel, sports, dining, and countless moments of joy. In alphabetical order, I thank Ahana, Aman, Amir, Andrea, Andreea, Ana Maria, Annika, Arabinda, Aris, Ashwani, Azalea, Bala, Bishwa,

Burcu, Cedric, Chris, Eirene, Eleni, Ellen, Felix, Filip, Fuyuan, Germano, Iason, Irmak, Ivan, Ivi, Kaushik, Khushraj, Kili, Kimaya, Klara, Leo, Lennard, Lia, Mahmoud, Mahdi, Maria, Mariam, Marin, Mehrdad, Mia, Michalis, Mohammad, Munko, Nastaran, Nina, Numair, Pascal, Pavel, Rajarshi, Ram, Richard, Ritam, Rosa, Sadegh, Sathiya, Satya, Simin, Simon, Srinidhi, Suhas, Stratis, Tung, Yugesh, Xuan. I would like to particularly mention Ashwani Anand, whose friendship made my PhD journey not only bearable but also joyful. His kindness and support to me and my family meant the world.

I am also eternally grateful to my parents; they made me who I am. Words cannot fully express my gratitude to them. My brothers Recep Tayyip and Yusuf Taha have been a constant source of love and their support has been my foundation. They will always remain my closest companions. I also extend my warmest thanks to other family members, friends, and well-wishers whose enthusiasm and belief in me have been a source of strength.

Last but certainly not least, I would like to express my most sincere apprecation to my family. Ezgi, *my wife, my sunshine, and my constant inspiration*, has always been by my side. She has borne countless challenges and hardships alongside me, and I could never have reached this point without her endless support. Every achievement is as much hers as it is mine, and I share all of my successes with her. My love for her grew stronger with each challenge we faced together on this journey. Our children, Kerim Vefa and Emir Ali, have always been a constant source of happiness. Pursuing my PhD while raising them was challenging, yet filled with joy. They taught me invaluable lessons in worldview, career planning, time management, and organization. I hope they will be proud of their dad. I dedicate this thesis to my children. If they choose to see it as an achievement, I hope it will inspire them in their future endeavors.

Contents

Chapter 1

Introduction

Machine Learning (ML) models are increasingly prevalent in various domains today, from software development Amazon-Q (2024); Github-Copilot (2024); Cursor (2024); Continue.dev (2024) to robotics Boston-Dynamics (2024); Amazon-Robotics (2022). As these ML models are integrated into safety-critical systems, it becomes crucial to assess their high degree of dependability. For instance, in self-driving cars, ML models control vehicle navigation, obstacle detection, and decision-making processes, directly impacting passenger and pedestrian safety Cui *et al.* (2019); Macrae (2022). Similarly, in aviation, these models help manage flight paths and control systems, where failures can result in catastrophic outcomes. These examples underscore the importance of validating machine-learning models' safety and reliability to perform intended functions without failure, and meet specified requirements under predefined conditions. Researchers have approached this problem from various aspects such as robustness or fairness of ML models 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)). However, beyond such specific properties, how does one specify, let alone check, general functional-correctness expectations from ML models? In this thesis, we introduce novel tools and approaches inspired by the well-established concepts and techniques from software testing to specify and fuzz ML artifacts for their functional correctness.

Given the breadth and complexity of the ML field, researchers bring diverse solutions to reliability of ML models from various perspectives and backgrounds Dalrymple *et al.* (2024); Khlaaf (2023). In our work, we adopt a software testing perspective to tackle these challenges. Software testing aims to identify bugs and validate that a program behaves as expected by running it with given inputs. Two main challenges in software testing are generating test inputs and finding test oracles. *Fuzzing* Godefroid (2020) is a popular method for generating test inputs, manifested by creating a large volume of mostly random test inputs and monitoring for unexpected behaviors such as crashes or other specification violations which may serve as test oracles. These specifications can take various forms, for instance, defining metamorphic relations between inputs and their

expected outputs, known as *metamorphic testing* Chen *et al.* (1998), has been found to be largely applicable and beneficial. Since software has become an essential part of our lives, these techniques and concepts have proven effective and crucial for validating software reliability. In this thesis, we tailor these techniques and concepts to assess the reliability of ML models.

One of the biggest recent advancements in machine learning has been in solving sequential decision-making problems Mnih *et al.* (2015); Silver *et al.* (2016, 2018); Issakkimuthu *et al.* (2018); Groshev *et al.* (2018); Garg *et al.* (2019); Toyer *et al.* (2020); Karia and Srivastava (2021); Schultz *et al.* (2024); Haarnoja *et al.* (2024), where an agent makes a series of decisions over time to achieve a specific goal. This often requires learning action policies, which are strategies or rules to determine the best action to take in each situation. In this thesis, we devise new techniques to test the reliability of these action policies. The most straightforward testing objective is to check if the policy leads to an undesirable outcome from a given state. However, sometimes such outcomes are unavoidable due to environment conditions. For instance, surrounding traffic could make it impossible to avoid a crash in autonomous driving or rough terrain could make it impossible for a robot to keep its balance. Therefore, we aim to answer the question: *how can we identify undesirable yet avoidable outcomes?* We present novel test oracles based on metamorphic relations between environment states and develop the $\pi-$fuzz framework utilizing them to identify bugs in action policies. $\pi-$fuzz takes as input a policy under test which controls the agent and an environment on which the agent is operating. It creates a diverse set of test states through its fuzzer component and determines bugs in the policy with the metamorphic oracles. We demonstrate the effectiveness of $\pi-$fuzz on diverse environments and policies.

Next, we formalize the metamorphic relations as k-safety properties, or *hyper*properties Clarkson and Schneider (2008), which describe relationships between multiple input-output pairs. Hyperproperties are specifications used in formal methods to specify complex requirements, such as information flow control, and consistency across executions. For example, a hyperproperty might specify that if a particular input leads to a certain output, then a similar input should lead to a similar output, ensuring consistency and reliability across multiple runs of the system. We define hyperproperties of ML models from various domains via metamorphic relations and show that hyperproperties can be generalized across various domains and ML models to specify their functional correctness. To express these hyperproperties, we create a declarative, domain-agnostic specification language called NOMOS[1]. NOMOS helps define the general functional correctness of ML models and comes with an automated testing framework. 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 test cases for which the model violates the specified properties.

[1] Law in Greek

We demonstrate its effectiveness in finding property violations across various ML application domains, such as tabular data, image classification, sentiment analysis, and speech recognition.

As a next step, we extend the NOMOS syntax and scope to accommodate code translation models, where the task is to take a program written in one programming language and re-write it in another. The most desirable property of a code translation model to re-write programs by preserving input/output behavior or in other words, semantic equivalence. However, testing this property is hard for arbitrary code such as incomplete code fragments and programs with complex types, since both input generation and output monitoring can be challenging. We build on NOMOS to enable it to express and validate a wide range of hyperproperties, ranging from purely syntactic to purely semantic. An example 2-safety syntactic property reads as *Given input function i_1, input function i_2 is created by renaming a random parameter in i_1. The arity of their corresponding output functions should be equal.* We evaluate multiple state-of-the-art code translation models against 38 properties and present violations.

To conclude, in this thesis, we tackle the critical issue of validating the reliability and safety of ML models integrated into numerous systems in our daily lives. Our approach draws from well-established software testing techniques, tailoring them to the unique challenges posed by ML models. By introducing the π-fuzz framework and NOMOS specification language, we provide powerful tools for generating test cases and identifying functional correctness issues. Overall, this thesis makes contributions to validating the reliable use of ML models in safety-critical systems. The methodologies and tools developed throughout this thesis provide a foundation for future research for safety and reliability of ML systems from software testing perspective. In the rest of this chapter, we provide summaries of the main contributions of this thesis in three sections.

1.1 π-fuzz: Fuzz Testing of Action-Policies

Recently, the use of neural networks (NN) to learn action policies has shown great success in sequential decision-making problems such as game playing or robot task planning. Action policies can make real-time decisions in dynamic environments by evaluating the current state to determine the next action. However, potential policy *bugs* or undesirable behaviors, are a concern. Testing is a natural paradigm to assess the quality and reliability of these policies. In this chapter, first, we study what constitutes a *bug* in this context. Afterwards, we present novel test oracles to identify whether the policy has a bug manifested on a given state. Assessing the reliability of a policy requires querying oracles with a wealth of diverse test states. For this, we create a test-state generation, or a fuzzing, component. We implement novel test oracles and the fuzzer in a tool called π-fuzz.

Previous work on testing sequential decision-making systems often assumes that correctly designed policies can always avoid failure conditions. However, this is not always true, as some failures, such as unavoidable traffic collisions in autonomous driving, may be inherent to the environment and not due to policy flaws. To determine if a failure is a policy bug, one can compare the current policy π with an optimal policy π^*: if π^* could avoid the failure from the same state, the failure reflects a bug in π. Two types of policy bugs are defined: *seed-bugs*, specific to deterministic behaviors tied to a random seed, and *bugs*, which arise when π has a higher failure probability compared to π^*. While seed-bugs approximate bugs in probabilistic environments, they coincide in deterministic scenarios.

Searching for optimal policies to detect bugs is computationally expensive, much like lacking knowledge of the correct output in software testing. Inspired by software metamorphic testing, sequential decision-making employs *relaxations*, which define relationships between states. For example, in autonomous driving, a state with less traffic is considered a relaxation of a state with more traffic. If a policy π that succeeds in the relaxed state fails more frequently in the original state, it exhibits a bug. Relaxations enable the specification of metamorphic relations and serve as test oracles for sequential decision-making, focusing on overall outcomes (solving or failing) rather than immediate actions.

Using relaxation relations, we develop practical oracles which are covered in full details in Chapter 2 to detect *bugs* and *seed-bugs* in action policies. These oracles leverage comparisons between states and their relaxed counterparts to identify failures attributable to policy flaws. For test-state generation, we draw inspiration from fuzzing, which randomly mutates program inputs to maximize diversity. In our setting, mutations correspond to random actions, and diversity is measured by the Euclidean distance between states. The fuzzer generates a pool of diverse states by performing random walks and

retaining only those mutations that significantly differ from existing states, ensuring a varied set of test cases. We implemented these techniques in a framework called π-fuzz, a generic, environment-agnostic policy testing tool, which takes as input a policy under test π and an environment E.

We evaluate π-fuzz on three single-agent games called Highway, LunarLander, and BipedalWalker, with policies learned by reinforcement learning. Overall, we show that **metamorphic oracles are highly useful for identifying bugs in action policies**, and they are superior to the rule-based and failure-based alternatives we evaluate. Also, our fuzzer implementation effectively explores a diverse range of states, which is crucial for ensuring that a testing campaign provides comprehensive and representative coverage.

1.2 Specifying and Testing Safety Properties of ML Models with NOMOS

In the previous section, we presented how metamorphic relations can be formed to obtain test oracles specifically for action policies. Here, we study the generalization of metamorphic relations across domains and ML models in the form of *hyper*properties, also known as k-safety properties, which are specifications used in formal methods, expressing functional-correctness properties by reasoning about k different executions. For example, 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–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 an individual model execution. We show the wide applicability of k-safety properties for machine-learning models and present the first specification language, NOMOS, for expressing them. We also operationalize the language in a framework for automatically validating such properties using metamorphic testing.

We demonstrate the wide applicability of general, user-provided k-safety properties for ML models. In practice, such properties are defined by users, thus expressing model expectations that are deemed important in their particular usage scenario. NOMOS allows a user to specify properties such as the above ones of an ML model under test. Figure 1.1 demonstrates an example property of an action policy written in NOMOS. This property illustrates the idea of BUGORACLE relying on relaxations in LunarLander domain we defined in earlier section. 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. We go over this NOMOS property in more detail in Chapter 3.

```
1   input s1;
2   var s2 := relax(s1);
3   output o1;
4   output o2;
5   {
6     o1, o2 = 0, 0
7     for _ in range(10):
8       rs = randInt(0, MAX_INT)
9       o1 += play(s1, rs)
10      o2 += play(s2, rs)
11  } # 0-lose, 1-win
12  ensures o1 <= o2;
```

Figure 1.1: LunarLander 20-safety property

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 Figure 1.1, line 2 describe the transformation to input `s1` in order to obtain `s2`, and line 12 specifies the relation between the corresponding outputs. We, therefore, design a framework for validating a model against a NOMOS specification using metamorphic testing. This framework compiles the NOMOS specification into a test harness, i.e., a program that tests the model against the specified properties. Our implementation parses NOMOS specifications and after semantically checking the parsed abstract syntax tree (AST), our framework translates the AST into the Python program constituting the test harness. The test harness employs a test generator and an oracle component, for generating inputs to the model using metamorphic testing and for detecting post-condition violations, respectively. To present the effectiveness of our testing framework, we trained models using seven datasets from five application domains, namely, tabular data (COMPAS and GermanCredit), images (MNIST), speech (SpeechCommand), natural language (HotelReview) and action policies (LunarLander and BipedalWalker). Our framework was able to find violations in all of the models. Most violations were exhibited through tens or hundreds of unique tests. This demonstrates that our framework is effective in detecting bugs. Further details regarding the domains and models are covered in Section 3.4.

1.3 Automatically Testing Functional Properties of Code Translation Models

Large language models (LLMs) are becoming increasingly practical for translating code across programming languages. Even though automated code translation significantly boosts developer productivity, a key concern is whether the generated code is correct. Existing work initially used manually crafted test suites to test the translations of a small

corpus of programs Rozière *et al.* (2020); these test suites were later automated Rozière *et al.* (2022). We devise an approach for automated, functional, property-based testing of code translation models. To this end, we build on NOMOS Christakis *et al.* (2023) to enable it to express and validate a wide range of k-safety properties about code translation models, ranging from purely syntactic to purely semantic properties of the translated code.

We extend NOMOS to support code translation models for expressing and testing their properties. First, we incorporated two new classes of domain-specific functions, namely, *program-transformation* and *program-inspection* functions. Intuitively, transformation functions mutate programs in a controlled manner and inspection functions infer program features. For instance, transformation function `addConditional` adds a random conditional in a given function without affecting its input/output behavior, and inspection function `numConditionals`, returns the number of conditionals in a given function. In total, we added 7 transformation and 5 inspection functions to express our properties.

Consider the syntactic 1-safety property shown in Figure 1.2a expressing that, when translating Java code into C++, the number of conditionals in the input (Java) program should match the number of conditionals in the output (C++) program. On the other hand a syntactic 2-safety property about a Java-to-Python translation is shown in 1.2b. On line 1, the property declares an input program `pj1`. Unlike the syntactic 1-safety property, it also declares an additional program `pj2` on line 2—`pj2` is generated by adding a random conditional to `pj1` such that the input/output behavior of the code remains unaffected. For instance, the body of the conditional may just consist of a print statement. The postcondition checks that the number of loops in `pp1` and `pp2` matches.

```
1 input pj;
2 output pc;
3 {
4   pc = transpile(pj, "java", "cpp")
5 }
6 ensures numConditionals(pj, "java") == numConditionals(pc, "cpp");
```

(a) Syntactic 1-safety property.

```
1 input pj1;
2 var pj2 := addConditional(pj1, "java");
3 output pp1;
4 output pp2;
5 {
6  pp1 = transpile(pj1, "java", "py")
7  pp2 = transpile(pj2, "java", "py")
8 }
9 ensures numLoops(pp1, "py") == numLoops(pp2, "py");
```

(b) Syntactic 2-safety property.

Figure 1.2: Example *k*-safety specifications for code translation models.

We also go a step further and explore the usage scenario where a user simply aims to obtain a correct translation of some code with respect to certain properties without necessarily being concerned about the overall quality of the model. We come up with a search procedure that repeatedly invokes the testing procedure with mutated model parameters to obtain less number of violated properties. It returns the model output as soon as all properties are satisfied by the current model instance. If all properties cannot be satisfied within a given search budget, it returns the best model output, which results in the fewest violated properties. We dive in details of this method in Section 4.3.3.

We evaluate our testing procedure on pre-trained models TRANSCODER Rozière *et al.* (2020), DOBF Lachaux *et al.* (2021), TRANSCODER-IR Szafraniec *et al.* (2023), and STARCODER Li *et al.* (2023a). We find out that our testing procedure is highly effective at detecting violated properties, with most issues identified using a beam size of 1. Increasing the beam size to 3 reduces the number of violations, indicating improved translation quality. Additionally, our search procedure significantly increases the number of passing programs and nearly halves the violated properties. Therefore, it further improves the overall translation quality with respect to the number of violated properties.

1.4 Outline and Publication Details

This dissertation is based on publications of which I am the main author during my Ph.D. studies. The list below shows each chapter and its related publication.

[1] Chapter 2 presents π-fuzz, a fuzzing framework for action policies. π-fuzz involves various metamorphic test oracles for identifying bugs in action policies. This work was published in ISSTA'22 under the title Metamorphic Relations via Relaxations: An Approach to Obtain Oracles for Action-Policy Testing Eniser *et al.* (2022). In this work, my co-authors and I came up with the novel idea presented in this paper. I implemented this idea and released it as an open-source solution. I selected appropriate benchmarks, set up the evaluation environment, and conducted all experiments to collect results. I wrote the sections explaining the main contributions and created the figures, algorithms, plots and tables, which were later revised by my co-authors.

[2] Chapter 3 presents NOMOS, a domain-specific language for specifying hyperproperties of machine learning models. NOMOS comes with a built-in testing framework for validating user specified properties in machine learning models. We present NOMOS' application on various domains such as image and speech classification and sentiment analysis. This work was published in IJCAI'23 under the title Specifying and Testing k-Safety Properties for Machine-Learning Models Christakis *et al.* (2023). In this work, my co-authors and I proposed creating NOMOS domain-specific language. My advisor, our collaborator, and I refined its grammar through multiple iterations. I implemented the grammar and the complete testing framework, set up the evaluation, and conducted all experiments. I wrote the sections on main contributions and added all experimental data, which my co-authors later revised and refined.

[3] Chapter 4 presents extension of NOMOS to code translation domain. We devise novel relevant properties using NOMOS and test various open-source models against them. This work was published in AAAI'24 under the title Automatically Testing Functional Properties of Code Translation Models Eniser *et al.* (2024a). My co-authors and I came up with the main idea behind applying NOMOS to the code translation use case. My advisor, our collaborator, and I improved and made it more comprehensive through multiple discussions. I extended the implementation and conducted the experiments. I contributed to writing the paper by explaining the main idea and presenting the experimental evaluation.

Chapter 2

π-fuzz: Fuzz Testing of Action Policies

Testing is a promising way to gain trust in a learned action policy π, in particular if π is a neural network. A "bug" in this context constitutes undesirable or fatal policy behavior, *e.g.*, satisfying a failure condition. But how do we distinguish whether such behavior is due to bad policy decisions, or whether it is actually unavoidable under the given circumstances? This requires knowledge about optimal solutions, which defeats the scalability of testing. Related problems occur in software testing when the correct program output is not known.

Metamorphic testing addresses this issue through metamorphic relations, specifying how a given change to the input should affect the output, thus providing an oracle for the correct output. Yet, how do we obtain such metamorphic relations for action policies? Here, we show that the well explored concept of relaxations in the machine learning community can serve this purpose. In particular, if state s' is a relaxation of state s, *i.e.*, s' is easier to solve than s, and π fails on easier s' but does not fail on harder s, then we know that π contains a bug manifested on s'.

We contribute the first exploration of this idea in the context of failure testing of neural network policies π learned by reinforcement learning in simulated environments. We design fuzzing strategies for test-case generation as well as metamorphic oracles leveraging simple, manually designed relaxations. In experiments on three single-agent games, our technology is able to effectively identify true bugs, *i.e.*, avoidable failures of π.

2.1 Introduction

Action policies represented by neural networks are highly successful in complex, sequential decision making problems, specifically in games Mnih *et al.* (2015); Silver *et al.* (2016, 2018) and increasingly in Artificial Intelligence (AI) Planning Issakkimuthu *et al.* (2018); Groshev *et al.* (2018); Garg *et al.* (2019); Toyer *et al.* (2020); Karia and Srivastava (2021); Ståhlberg *et al.* (2022). Once a policy π has been learned, it can be used to make real-time decisions in dynamic environments, simply by calling $\pi(s)$ on the current state s to obtain the next action. This approach, however, comes with obvious safety con-

cerns due to potential policy bugs, *i.e.*, undesirable or even fatal policy behavior. Testing is a natural paradigm, given its scalability, to address these concerns.

But what is a "bug" in this context? In many environments, undesirable or fatal behavior can be unavoidable – *e.g.*, traffic making it impossible to avoid a crash in autonomous driving, or a state in which it is impossible for a bipedal robot to keep its balance. Such situations are not bugs in π as the bad behavior is not actually due to bad policy decisions. In general, in order to know whether a situation constitutes a bug in π, we need to know the optimal policies, which minimize the probability of failure. This would defeat the scalability of testing.

Prior work on testing in the sequential decision making domain does not address this issue. It considers a "system" that takes decisions in an environment and tries to find situations where a failure condition ϕ is satisfied (*e.g.*, Dreossi *et al.* (2015); Akazaki *et al.* (2018); Koren *et al.* (2018); Ernst *et al.* (2019); Lee *et al.* (2020), see Corso *et al.* (2021) for an overview). Such an approach implicitly assumes that a correctly designed system – in our case, a learned action policy – can always avoid ϕ. This shortcoming was recently pointed out by Steinmetz et al. Steinmetz *et al.* (2021), who analyzed the possibility of identifying sub-optimal policy behavior through upper- and lower-bounding techniques. Here, we instead take inspiration from software testing, providing an alternative technique to detect avoidable failures.

Not knowing the optimal policies is akin to not knowing the correct output of a program. Metamorphic testing Chen *et al.* (1998) addresses the latter by testing program behavior on inputs chosen such that it is known how the respective outputs should relate. That is, one specifies a *metamorphic relation*, encompassing a relation R^I over inputs together with a corresponding necessary relation R^O over outputs. If, for inputs i, i' with $R^I(i, i')$, the outputs o, o' do not satisfy $R^O(o, o')$, then we know there is a bug. Thus, R^O provides a test *oracle* for the correct output.

However, this oracle still requires knowledge about correct outputs in the form of R^O. For example, action decisions in autonomous driving can be tested using metamorphic relations derived from well known human-designed rules, such as "speed down by 25% if rainy" Zhang *et al.* (2018); Tian *et al.* (2018); Deng *et al.* (2020); Luu *et al.* (2024). But how do we come by metamorphic relations in general, sequential decision making problems?

Our approach We answer this question in terms of a relation R^O, not over specific output actions a vs. a' of a policy, but over *the space of solutions below states s vs. s'*. Such relations are very common and well explored in AI, namely in the form of over-approximations obtained through *relaxation*. More specifically, assume that states

s and s' are related in terms of a relaxation relation $R(s, s')$, identifying that s' is easier to solve than s. If π fails on easier s' but not on harder s, then we know that π contains a bug manifested on s'. This is our key insight: *relaxations provide a means to specify metamorphic relations, and thus, a test oracle in general, sequential decision making problems*. Furthermore, this approach captures sequential policy behavior, rather than merely immediate outputs as in all other works on metamorphic testing.

As indicated by our notation above, we focus on *state relaxations* R, which modify only the state and not any other aspects of the agent's task. Such relaxations are often quite natural and easy to obtain. For example, when obstacles need to be avoided, states can be relaxed by removing obstacles; when resources are limited, relaxations can increase resource availability; when there are time constraints, these constraints can be relaxed (*e.g.*, by postponing a deadline).

In this thesis, we do not yet investigate the automatic generation of such relaxations. Instead, we perform case studies in three 2D-world single-agent games involving (fixed or moving) obstacles, and manually design relaxation relations $R(s, s')$, where s' has easier-to-avoid obstacles. It is important to note that, while this involves manual per-domain labor, it requires hardly any domain *knowledge* – relaxing obstacles is trivial. This is in stark contrast to knowing the difference between the optimal solutions for s and s', which constitutes the kind of knowledge we would need in order to design a traditional metamorphic output relation R^O.

Our testing framework addresses Markov decision processes (MDPs), of which we require access only to a simulator (given state s and action a, output an outcome state s'). For test-state generation, we take inspiration from fuzzing Miller *et al.* (1990); AFL (2024); Lib (2024), which mutates program inputs randomly with a bias to maximize diversity. We transfer this idea to our setting by taking input mutations to be random action applications, and measuring test-state diversity in terms of Euclidean distance.

We implemented our techniques in a framework we call π-fuzz. We evaluate π-fuzz on three single-agent games, with policies learned by reinforcement learning. Our experiments show that fuzzing is effective in generating a diverse set of states, and that our metamorphic oracles are able to identify thousands of unique bugs even in well trained policies. (The source code of π-fuzz and all data necessary to reproduce our experiments are available at `https://github.com/Practical-Formal-Methods/pi-fuzz`.)

Contributions In short, we make the following contributions:

- We introduce the first technique for metamorphic testing of action policies. Our key insight is to use the concept of relaxations to design novel oracles that do not require optimal policies.

- We propose the π-fuzz fuzzing framework, which combines a fuzzer for action policies with metamorphic test oracles.
- We evaluate π-fuzz on three single-agent games, with policies learned by reinforcement learning.

Outline The rest of this chapter is organized as follows. Section 2.2 reviews related work. Section 2.3 provides background on the general setting and introduces notation. Section 2.4 gives an overview of π-fuzz. In Section 2.5, we formally define "policy bugs", and in Section 2.6, we describe our metamorphic test oracles. In Section 2.7, we describe our fuzzing component before introducing our three case studies in Section 2.8. Finally, we evaluate π-fuzz in Section 2.9 and conclude in Section 2.10.

2.2 Related Work

Testing neural networks Testing neural networks is a well studied field in the literature. However, most of this work focuses only on a single invocation of a neural network, for example, in the case of image classifiers. For instance, there are several techniques that introduce test coverage criteria at the neural-network level Pei *et al.* (2017); Kim *et al.* (2019); Gerasimou *et al.* (2020); Ma *et al.* (2018) for more efficient testing. Sun et al. Sun *et al.* (2018) devise a technique for concolic testing of neural networks. Wang et al. Wang *et al.* (2019) detect adversarial inputs by using ideas from mutation testing. Finally, there are some existing testing techniques that use metamorphic relations in specific domains, such as automated driving Zhang *et al.* (2018); Tian *et al.* (2018); Deng *et al.* (2020), object detection Zhou and Sun (2019), and translation He *et al.* (2020).

However, applying such existing work to test reinforcement learning (RL) policies is non-trivial since such policies are typically assessed by the total reward at the end of a sequence of decisions (i.e., *multiple* network invocations) that are taken from a given initial state. Differentially testing the optimality of each decision in the sequence is impractical in many real-world domains since it would require access to an optimal policy. Apart from that, the aforementioned approaches for metamorphic testing of autonomous driving decisions Zhang *et al.* (2018); Tian *et al.* (2018); Deng *et al.* (2020); Luu *et al.* (2024) are domain specific and leverage human knowledge in the form of driving rules – knowledge about solutions that is not available in general.

On the other hand, our approach only relies on relaxation relations, which are typically straightforward and apply to a wide range of domains. To nevertheless provide an empirical comparison to rule-based approaches, we experiment with simple "change nothing" rules. These define metamorphic relations prescribing that, if s' is a relaxation of s, then what works for harder s should also work for easier s' – the policy should not change.

However, this may inaccurately classify s' as a bug, leading to false positives (lack of *precision*). Further, our experiments show that this method has lower *recall* (more false negatives) than ours.

Bug finding for decision making policies There is recent work on finding falsifying inputs for RL-based decision making policies in hybrid systems Dreossi *et al.* (2015); Akazaki *et al.* (2018); Koren *et al.* (2018); Ernst *et al.* (2019); Lee *et al.* (2020); Corso *et al.* (2021). Unlike π-fuzz, these techniques do not address scenarios where the policy is expected to fail (e.g., unavoidable crash state). As mentioned earlier, Steinmetz et al. Steinmetz *et al.* (2021) recently pointed this out and used upper- and lower-bounding techniques to identify sub-optimal policy behavior. In contrast, our work takes inspiration from metamorphic testing to ensure that the identified failures are avoidable.

Another study Pang *et al.* (2021) focuses on testing deep NN models solving MDPs via techniques such as reinforcement learning and imitation learning Ho and Ermon (2016). They develop a fuzzer that is guided by a heuristic novelty measure to detect crash triggering initial states. There are three key differences with our work: (1) Their state mutations employ hand-crafted bounds to suppress bugs due to unavoidable crash states. Configuring these bounds can be difficult, if not impossible, for complex domains. In contrast, our approach by construction never reports bugs due to unavoidable crashes. (2) Bounds that limit the magnitude of each mutation may result in insufficient exploration of the input space. In π-fuzz, such bounds are not necessary. (3) The aforementioned work can identify only crash-triggering states. In contrast, our approach can easily be generalized to identify states on which the reward is less than expected.

Another line of work inspires from software testing and verification practices for finding bugs in action policies. Tappler *et al.* (2022) employ search-based testing, while Biagiola and Tonella (2024) use surrogate models for validation. Vu *et al.* (2024) integrate ProB Leuschel and Butler (2003), a formal method tool, for agent validation and safety shield testing. Mazouni *et al.* (2024) investigate fault diversity, and He *et al.* (2024) adopt a curiosity-driven approach for sequential decision-making testing. Varshosaz *et al.* (2023) discuss formal specification and testing, and Li *et al.* (2023b) leverage generative models for decision-making policy testing. Similar to our work, Mazouni *et al.* (2025) explores metamorphic testing for testing AI planning agents. Eisenhut *et al.* (2023, 2024) build upon our work, where the former explores automatic relaxation relation generation and the latter develops a biased fuzzer for finding bug states.

Safe reinforcement learning There exists a large body of work on safe reinforcement learning. On a high level, Safe reinforcement learning aims to ensure that agents operate within safe bounds (see Garcia et al. García and Fernández (2015) for an overview).

Bastani *et al.* (2018) introduce a method for verifiable reinforcement learning through policy extraction, enabling formal verification of learned policies to ensure correctness and safety. Hunt *et al.* (2021) propose a framework for verifiably safe exploration in reinforcement learning, focusing on ensuring safety guarantees during the learning process for end-to-end systems. Alshiekh *et al.* (2018) introduce a shielding approach to prevent unsafe actions. Könighofer *et al.* (2020) synthesize shields to guide RL agents safely, while Carr *et al.* (2023) extend this to partial observability scenarios. Yang *et al.* (2023) utilize probabilistic logic shields for safety, and Könighofer *et al.* (2023) focus on online shielding to adaptively ensure safe behavior during learning. While safe RL and policy testing share the same goals, their methods are fundamentally different. However, we can imagine interesting combinations as part of our future work; for instance, by feeding detected bugs back into policy re-training, testing may become part of a larger safe-RL loop.

2.3 Context and Notations

Our methods address discrete-time Markov decision processes as follows. An MDP is a tuple $M = (S, A, T, S_0)$ of **states** S; **actions** A; **transition probability function** $T : S \times A \mapsto \mathcal{D}(S)$, where $\mathcal{D}(S)$ denotes the set of probability distributions over S; and **initial states** $S_0 \subseteq S$ (of which one $s_0 \in S_0$ will be chosen randomly at execution time).

A **policy**, also **agent**, is a function $\pi : S \mapsto A$ that chooses actions in S. We consider policies π represented by neural networks. The policy is typically trained to maximize rewards associated with states or state transitions. Our approach is agnostic to how this is done. A **run** of a policy π on an (arbitrary) state $s_0 \in S$ is a state/action sequence $\sigma = \langle s_0, a_0, s_1, a_1, \ldots \rangle$, where, for all i, $a_i = \pi(s_i)$ and $\mu(s_{i+1}) > 0$, where $\mu = T(s_i, a_i)$ and $\mu(s_{i+1})$ denotes the probability of reaching state s_{i+1} from s_i by taking action a_i.

Note that this definition of policy is restricted in terms of being memoryless and deterministic. Both restrictions can be lifted in principle, but deterministic memoryless policies are relevant in their own right and form a natural starting point for the investigation of metamorphic action-policy testing. We assume that π is represented as an NN classifier whose final layer can also be interpreted as a probability distribution $\hat{\pi}(s) \in \mathcal{D}(A)$ over actions. We make use of the latter in fuzzing, by sampling $\hat{\pi}(s)$ in order to explore MDPs in which random actions do not lead to interesting states.

We assume a given non-temporal **failure condition** ϕ that should be avoided by the agent – exploring our approach for temporal ϕ remains a topic for future work. We say that a run σ **fails** if there exists s_i along σ such that $s_i \models \phi$; otherwise, we say that σ **succeeds**. We denote by $P_\phi(\pi, s)$ the probability that the run of π on s fails, and by $P^*_\phi(s)$ the minimal such probability achieved by any policy.

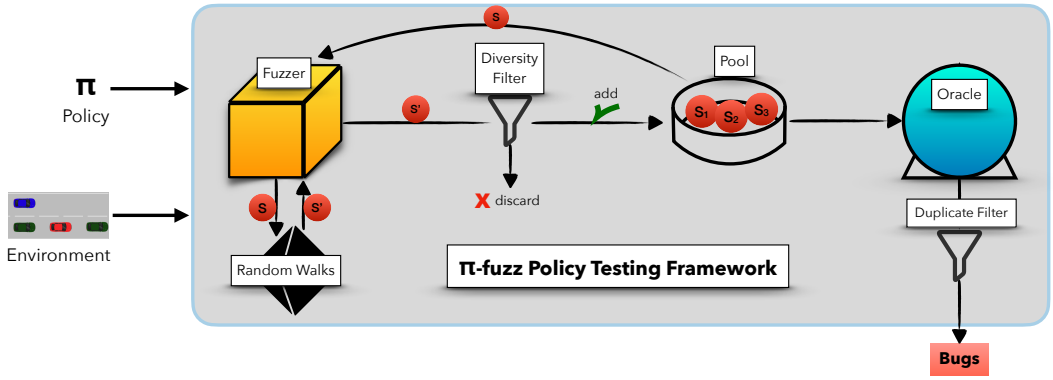


Figure 2.1: Overview of π-fuzz framework.

We do not assume that we have a declarative model of M; a simulator suffices to apply our methods. We merely assume that the representation of states is state-variable based, *i.e.*, each s is uniquely identified by a value assignment to a vector of **state variables** $(v_1, \ldots, v_n)$. The domains of the state variables do not matter to our approach as long as Euclidean distance can be defined (needed in our state-diversity notion). For simplicity, we assume that the state variables are real-valued in this chapter, *i.e.*, states s map each v_i to $\mathbb{R}$.

We refer to the simulator as the **environment**, denoted E. It provides the programmatic interfaces E**.randomInit**(), which returns a random initial state $s_0 \in S_0$; E**.setState**(s), which sets the environment state to $s \in S$; and E**.step**(s,a), which, given $s \in S$ and $a \in A$, picks and outputs a state s' according to the distribution $T(s,a)$.

We furthermore assume that E has a parameter ρ – the random seed – and an interface E**.setSeed**(r) setting $\rho := r$. We use this interface in part of our methodology to fix specific environment behaviors and identify bugs pertaining to those. Namely, whenever we check whether π contains a bug manifested below a state s, we call E.setSeed(r) directly before running π on s, determinizing T as a function of state, action, and run length so far. We denote the resulting unique **run of** π **on** s **given random seed** r by $E^r.\sigma[\pi, s]$.

2.4 π-fuzz Policy Fuzzing Framework

Figure 2.1 provides a high-level overview of our π-fuzz policy-testing framework. As shown in the figure, π-fuzz takes as input a policy under test π and an environment E. The framework consists of two main components: the **fuzzer**, which generates a diverse pool of test states s_i; and the **oracle**, which identifies policy bugs among these test states.

The fuzzer uses random walks to generate new states, which are then filtered by diversity to obtain the pool. We will describe our fuzzing algorithm in detail in Section 2.7. Our key contribution is the design of the oracle, via metamorphic relations based on relaxations. A duplicate filter at the end of this pipeline serves to provide unique bugs as the output of π-fuzz.

Our π-fuzz framework is implemented as a generic policy tester, independent of any specific environment (in fact, we use the same implementation across our case studies in this chapter). The input interface for π-fuzz consists of the neural network representation of π, in the PyTorch format; the aforementioned programmatic interface of E implementing E.randomInit(), E.setState(s), E.step(s,a) and E.setSeed(r); as well as a programmatic interface for metamorphic operations underlying the oracle, explained in Section 2.6.

2.5 Policy Bugs

In our context, we consider two notions of "policy bugs", one of which is specific to a fixed environment behavior, while the other quantifies over all possible such behaviors.

Definition 1 (Bug)**.** *Let $M = (S,A,T)$ be an MDP, E a simulator for M, π a policy, and $s \in S$ a state.*

*(i) We say that s is a **bug** in π if $P_\phi(\pi,s) > P^*_\phi(s)$.*

*(ii) Given a random seed $\rho = r$, we say that the run $E^r.\sigma[\pi,s]$ is a **seed-bug** in π if $E^r.\sigma[\pi,s]$ fails, but there exists a policy π' such that $E^r.\sigma[\pi',s]$ succeeds.*

Bugs (i) arguably capture the canonical understanding of policy bugs when testing for failure avoidance in a probabilistic environment. Seed-bugs (ii) are an approximation that allows to consider individual environment behaviors. If π fails but π' succeeds given the same random seed, then this indicates that π is faulty. This is, however, not necessarily the case: (ii) does *not* in general imply (i) because the decisions causing failure on r may be beneficial for other environment behaviors. Hence, (ii) is merely a pragmatical proxy for (i). That said, (i) and (ii) coincide for deterministic environments; and in our case studies, most states s with seed-bugs found by our metamorphic oracles are in fact bugs. Moreover, (ii) is much faster to evaluate than (i), which makes it useful for practical purposes.

Note that seed-bugs are best characterized by the actual run $E^r.\sigma[\pi,s]$, rather than state s alone, as there can be many different environment behaviors below s and only some of them may exhibit the observed failure.

2.6 Metamorphic Oracles

In this section, we explain the principle of relaxation-based metamorphic oracles, specify the oracles used in our case studies, and discuss how suitable relaxations may be obtained in general, especially when considering the relaxation literature.

2.6.1 Metamorphic Oracles via Relaxation

Obviously, Definition 1 cannot be tested efficiently on large state spaces. We adapt the idea of metamorphic testing to solve this issue. The key element is a state relaxation:

Definition 2 (State Relaxation)**.** *Let $M = (S, A, T, S_0)$ be an MDP, and E a simulator for M. We say that $t \in S$ **relaxes** $s \in S$ if, for every policy π_s, there exists a policy π_t such that, for every random seed $\rho = r$, whenever $E^r.\sigma[\pi_s, s]$ succeeds, then $E^r.\sigma[\pi_t, t]$ also succeeds.*

*We say that $R \subseteq S \times S$ is a (state) **relaxation** if, for every $(s,t) \in R$, t relaxes s.*

We will illustrate and discuss this definition below. In a nutshell, a relaxed state t allows to adapt any policy for s to achieve the same (or more) failure-avoidance ability. In that sense, intuitively, "t is easier to solve than s". Definition 2 captures the most general condition under which this is the case, and where our metamorphic oracles hence work as intended.

Namely, the idea is quite simple: if t is easier to solve than s, but the policy π is worse on t than on s, then π's behavior on t must be wrong:

Proposition 3 (Metamorphic Oracle)**.** *Let $M = (S, A, T, S_0)$ be an MDP, E a simulator for M, and R a relaxation. Let $s, t \in S$ be states such that $(s,t) \in R$. We have:*

(i) If $P_\phi(\pi, s) < P_\phi(\pi, t)$, then t is a bug in π.

(ii) If $E^r.\sigma[\pi, s]$ succeeds but $E^r.\sigma[\pi, t]$ fails, then $E^r.\sigma[\pi, t]$ is a seed-bug in π.

Proof. We prove each part of the proposition as follows:

(i) First, note that we have $P^*_\phi(s) \geq P^*_\phi(t)$: Given a policy π^*_s that minimizes the probability of failure on s, by Definition 2 there exists a policy π_t for t that succeeds in all cases where π^*_s does. Hence, $P_\phi(\pi^*_s, s) \geq P_\phi(\pi_t, t)$, showing that $P_\phi(\pi^*_s, s) \geq P_\phi(\pi^*_t, t)$ for any policy π^*_t that minimizes the probability of failure on t. Together with prerequisite (i), we get $P_\phi(\pi, t) > P_\phi(\pi, s) \geq P^*_\phi(s) \geq P^*_\phi(t)$, which shows the claim.

(ii) By prerequisite, the run of π on s given random seed r succeeds, and $(s,t) \in R$. Hence, by Definition 2, there exists a policy π_t for which the run on t given r succeeds. As the latter is not the case for π, $E^r.\sigma[\pi, t]$ is a seed-bug in π.

□

To illustrate Definition 2 and the kind of practical relaxations we employ in our experiments, let us briefly consider the case studies we contribute. We experiment with three games (Highway, LunarLander, BipedalWalker) where an agent moves in a 2D-world and needs to reach a target position while avoiding (fixed or moving) obstacles. In each case, our relaxation relation R modifies the game landscape, making obstacles easier to avoid. Figure 2.2 (see Section 2.8) illustrates this. For each of the three games, the figure shows a state s on the left and a relaxed state t on the right.

Highway involves a car (red car in Figure 2.2a) navigating traffic on a 2-lane highway. Less traffic is easier to navigate. In the sense of Definition 2, whenever π_s manages to avoid crashing into traffic when started from s, we can achieve the same when starting from t simply by taking the same driving decisions; *i.e.*, the desired policy π_t behaves like π_s on the corresponding states. If the policy π under test takes different decisions on t, which crash more frequently, then π exhibits a bug on t. The other two games are similar in this regard: whenever π_s manages to avoid crashing on s, the same decisions avoid a crash on t, and so t relaxes s according to Definition 2.

A remarkable special case of Definition 2 is that of deterministic transitions, like in BipedalWalker where there is no noise in the effect of the robot's motor commands. In this case, the run of a policy from a state is unique, and t relaxes s if and only if either t is solvable (a succeeding policy exists), or s is unsolvable; in particular, all solvable states relax each other. This is very generous from a formal point of view, but it still makes perfect sense for bug detection: if s is solvable and $R(s,t)$, then we know that t is solvable. This is precisely the most general condition under which we can detect avoidable failures by comparing the behavior of π across states s,t: if $R(s,t)$ and π succeeds on s, then t is solvable, so failure of π on t constitutes a bug. Practical relaxation methods will, of course, instantiate the broad frame of Definition 2 with much more restrictive relations over states, like the relaxations in our case studies.

2.6.2 Metamorphic Oracles in our Case Studies

Given a state relaxation R as per Definition 2, Proposition 3 provides a tool to detect bugs and seed-bugs. We turn this into practical oracles for π-fuzz by sampling R a given number of times. Algorithm 1 specifies three different oracles along these lines. Let us discuss them from top to bottom.

The BUGORACLE algorithm checks whether a given test-pool state s_i generated by π-fuzz can be identified to be a bug. It does so by comparing $P_\phi(\pi, s_i)$ with $P_\phi(\pi, t_i)$ for *unrelaxed* states t_i, *i.e.*, harder states where $R(t_i, s_i)$. By Proposition 3, if the Boolean return value is 1, then s_i is a bug in π. Evaluating P_ϕ here is a sub-problem, and solving

Algorithm 1: Metamorphic oracles

```
1  Function BugOracle(R, π, s_i):
2      evaluate P_φ(π, s_i);
3      repeat ORACLE_BUDGET times
4          t_i = randomState({t_i | R(t_i, s_i)});
5          evaluate P_φ(π, t_i);
6          if P_φ(π, t_i) < P_φ(π, s_i) then
7              return 1
8      return 0;
9  Function BasicSeedBugOracle(R, π, s_i, r):
10     if E^r.σ[π, s_i] fails then
11         repeat ORACLE_BUDGET times
12             t_i = randomState({t_i | R(t_i, s_i)});
13             if E^r.σ[π, t_i] succeeds then
14                 return 1
15     return 0;
16 Function ExtSeedBugOracle(R, π, s_i, r):
17     B = {};
18     if E^r.σ[π, s_i] succeeds then
19         repeat ORACLE_BUDGET times
20             t_i = randomState({t_i | R(s_i, t_i)});
21             if E^r.σ[π, t_i] fails then
22                 B = B ∪ {E^r.σ[π, t_i]};
23     return B;
```

it precisely is intractable in itself for large state spaces. In our implementation, we approximate P_ϕ by sample runs.

The BasicSeedBugOracle algorithm proceeds in a similar manner, but checks for seed-bugs instead. The random seed r is an input to the oracle (set by the fuzzer, see Section 2.7) as the oracle's job is to identify bugs given a fixed environment behavior. The oracle returns 1 if π fails on the test state s_i but succeeds on one of the unrelaxed states t_i. By Proposition 3, $E^r.\sigma[\pi, s_i]$ is a seed-bug in this case.

Finally, ExtSeedBugOracle is an extension that applies in case π succeeds on the test state s_i given r. In this case, $E^r.\sigma[\pi, s_i]$ cannot be a seed-bug, but we may be able to identify relaxed states t_i as seed-bugs instead. The oracle leverages this possibility in the

obvious manner. In our case studies, many additional seed-bugs are found in this way. Note that we can define a similar extended version of BUGORACLE; however, such an oracle would be very slow.

To sample the relaxation relation R, our implementation in π-fuzz assumes that R is given in the form of a set of **metamorphic operations**: state-modification operators that either relax the given state (*e.g.*, by removing obstacles) or unrelax it (*e.g.*, by adding obstacles). The sampling then simply consists in applying a randomly chosen metamorphic operation with randomly chosen parameters (*e.g.*, which obstacles to remove, or where to add new obstacles).

Importantly, the magnitude of metamorphic operations affects oracle efficacy. If π works well on s_i and t_i is much easier, it is unlikely that the policy is bad on t_i, thus not leading to the detection of a bug. If π is bad on s_i and t_i is much harder than s_i, it is unlikely that the policy works well on t_i, again not leading to the detection of a bug. Therefore, in both directions, metamorphic operations should be applied cautiously, making small modifications only. Our operations modifying individual state attributes naturally support this. Also, for this reason, we do not chain over metamorphic operations, always applying only a single such operation when sampling R in Algorithm 1.

Section 2.8 outlines the metamorphic operations we use in each of our case studies. The algorithm parameter `ORACLE_BUDGET` is set to 500 in all our experiments.

2.6.3 Discussion: How To Obtain Relaxations

How can state relaxations for metamorphic oracles be obtained? In some special cases, relaxations modifying individual state attributes are actually quite easy to come by. Wherever obstacle avoidance plays a role, we can define metamorphic operations making obstacles easier to avoid, as in our case studies. Other simple examples include MDPs where resources, like fuel or energy, are consumed: we can then relax states by increasing the amounts of resources available. Similarly to this, if the MDP involves (discrete-time) deadlines by which something needs to be achieved, then states can be relaxed by postponing the deadlines. More generally, if some actions are possible only within given discrete-time time windows, then we can relax states by broadening those time windows. Note that such resource and time-constraint relaxations can potentially even be obtained fully automatically, as the metamorphic operations needed are generic (add resource/expand time-window border).

It remains, of course, an important question whether and how we can tap into the potential of research on relaxations in AI, where relaxations have been intensively investigated for the design of heuristic functions, *i.e.*, to compute lower bounds on goal distance (*e.g.*, Bonet and Geffner (2001); Edelkamp (2001); Helmert *et al.* (2014); Domshlak *et al.*

(2015)). The adaptation of these methods to our context is highly non-trivial as relaxations underlying heuristic functions make strong problem simplifications (*e.g.*, abstractions over-approximating transition behavior). This is in contrast to our need for cautious relaxation in small steps, as outlined above.

Another, perhaps more promising, source of state relaxations can be simulation relations Milner (1971); Gentilini *et al.* (2003), where $R(s,t)$ holds – t simulates s – iff for every outgoing transition of s there is a corresponding outgoing transition in t, leading to simulating outcome states $R(s',t')$. Such relations can potentially be extracted automatically if a declarative model of the MDP is available.

A recent work Eisenhut *et al.* (2023) has shown how to automatically design test oracles based on simulation relations between states, specifically in the context of planning. Two families of oracles have been proposed: state-morphing oracles and bound-maintenance oracles. State-morphing oracles generate morphed states from an input state and identify bugs based on the behavior of a policy π on these morphed states. The primary challenge here lies in generating meaningful morphed states. Bound-maintenance oracles, on the other hand, leverage quantitative simulation relations to propagate upper bounds on plan costs across encountered states, enhancing the evaluation of policy behavior.

2.7 Fuzzing Algorithm

We now discuss the fuzzer component from Figure 2.1 in more detail. The fuzzer builds up a pool of diverse states by relying on two sub-components, namely random walks and diversity analysis. Consider the pseudo-code in Algorithm 2.

First, the fuzzer adds a random initial state to the pool of states P (line 3). Until the fuzzer is interrupted (*e.g.*, via a user-provided time limit), it tries to incrementally expand P. To do so, it randomly decides (biased by a probability provided in parameter `INC_PROB` on line 5) to either select a random state from the pool (line 6) or select a new random initial state (line 8). Utilizing previously generated states (from the pool) allows us to explore the state space much more effectively as they are used as stepping stones to delve into unexplored territory. A random walk is then conducted on the resulting state s to obtain a new candidate state s' (line 9). If s' is sufficiently diverse, it is added to P (lines 10 – 11). Here, $d^{\text{Eucl}}(s,s')$ is the Euclidean distance between s and s', and `DIV_THRESH` sets a threshold for the minimum distance to the states already in pool P.

Once pool P is final, an oracle is called on each state $s_i \in P$ (lines 12 – 13). If the oracle requires a fixed random seed – like the two seed-bug oracles from Algorithm 1 – then the fuzzer chooses that seed here. Note that, to check for bugs in different environment behaviors, the oracle would need to be called with several seeds. Here, we show that

Algorithm 2: Fuzzing procedure

```
1  Function FUZZER(Env E, Policy π):
2      P = [];
3      P = ADD(E.RANDOMINIT(), P);
4      while ¬INTERRUPTED() do
5          if RANDOMBOOLEAN(INC_PROB) then
6              s = RANDOMSTATE(P);
7          else
8              s = E.RANDOMINIT();
9          s' = RANDOMWALK(E, π, s);
10         if min_{t∈P} d^Eucl(s', t) > DIV_THRESH then
11             P = ADD(s', P);
12     for each s_i ∈ P do
13         Run oracle on s_i (picking r if needed);
14 Function RANDOMWALK(Env E, Policy π, State s):
15     E.SETSTATE(s);
16     k = RANDOMINTRANGE(0, WALK_LENGTH);
17     if RANDOMBOOLEAN(POL_PROB) then
18         repeat k times
19             a = RANDOMPOLICYACTION(π̂(s));
20             s = E.STEP(s, a);
21     else
22         repeat k times
23             a = RANDOMACTION(E.actions);
24             s = E.STEP(s, a);
25     return s;
```

many seed-bugs can be found even when only trying a single seed for each s_i.

The diversity filter serves (similarly as in software testing) for higher confidence in π's ability to avoid failure, based on broad tests. Diversity of program inputs is typically facilitated by only adding inputs to the pool if they increase some form of code coverage (*e.g.*, statement, branch, or path coverage). Recently, several coverage criteria for NNs have emerged, such as neuron coverage Pei *et al.* (2017) in the context of NN robustness testing. In π-fuzz, we take inspiration from a coverage criterion Odena *et al.* (2019) based on the Euclidean distance between activation vectors for a given NN layer. Here, we apply this criterion to states – *i.e.*, the NN input vectors – instead.

The RANDOMWALK procedure conducts random walks in the usual manner, with one noteworthy design decision. Rather than always choosing actions uniformly at random (line 23), the algorithm sometimes samples the policy under test instead (line 19; recall that $\hat{\pi}(s) \in \mathcal{D}(A)$ interprets the final layer of the NN policy as a probability distribution over actions). The parameter `POL_PROB` (line 17) controls the trade-off between these two choices. Sampling the policy makes sense when random actions do not tend to lead to interesting states, *e.g.*, because states quickly become unsolvable. Indeed, as our empirical results show, this method yields advantages in all our case studies.

Regarding the parameters of Algorithm 2, in preliminary experiments we found that `INC_PROB` $= 0.8$ tends to work well across all of our case studies (for smaller values, exploration is insufficient), so we fix this parameter value. Similarly, we fix `POL_PROB` $= 0.2$ (for larger values, exploration is insufficient). For `DIV_THRESH` and `WALK_LENGTH`, good values depend on domain-specific aspects, *i.e.*, typical scale of state diversity, typical run length, typical level of risk incurred by long random walks. We, hence, fix specific values for each domain, listed as part of our case study descriptions in the next section. For `POL_PROB` and `DIV_THRESH`, interesting algorithm performance differences arise from setting them to 0 vs. > 0, so we evaluate these settings in our experiments.

2.8 Case Studies

We apply our π-fuzz framework to three case studies, called Highway, LunarLander, and BipedalWalker. Illustrations of their environments are shown in Figure 2.2. LunarLander and BipedalWalker are popular Gym Brockman *et al.* (2016) environments specialized for continuous control. We developed Highway as a new benchmark that simulates a simplified autonomous-driving task, navigating a highway through speed and lane changes in a way that avoids collisions with traffic. In all these case studies, the failure condition ϕ is given in terms of a specific environment state in which the agent ends up when it crashes into an obstacle. All agents were trained on a Debian 10 machine with 768 GB of memory, 32 CPUs (Intel(R) Xeon(R) Gold 6134M), and 2 GPUs (V100 Nvidia Tesla

with 32 GB of memory).

We next describe each case study, including the domain itself, the algorithms and parameters we used for learning the policy π, the metamorphic operations for the oracle, and the domain-specific settings of `DIV_THRESH` and `WALK_LENGTH`.

2.8.1 Highway

The Highway domain consists of a two-lane, finite-length street. The left lane is for *speed maniacs*, who are relatively fast, and the right lane is for *safety freaks*, who are slow. Neither of these actors may change lanes, and their speed is constant. The agent appears at the beginning of the street, and the task is to reach the end without crashing into other cars. There are five discrete actions: switch lane to right or left, speed up, slow down, noop. Other cars may enter or leave the highway stochastically while the agent is moving. In case of a crash, the game ends immediately with a reward of -100 points. Reaching the end of the street is rewarded with $+100$ points. The discount factor is $\gamma = 0.95$, incentivizing the agent to drive fast. Given the road length, the best-case achievable reward is ca. 30.

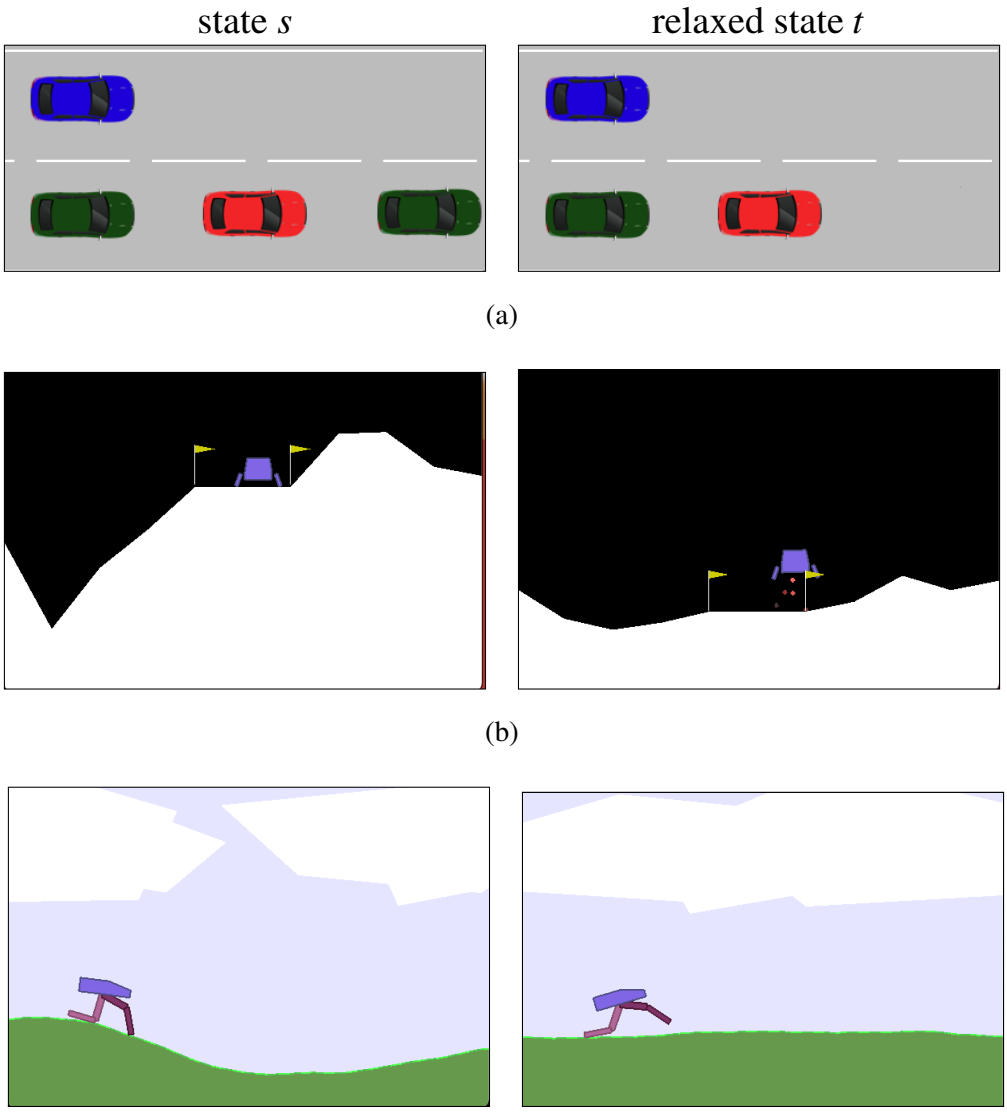
Policy training We train the agent using our own implementation of DQN Mnih *et al.* (2015). It is well trained, achieving an average reward of 21 points (after 20000 training episodes).

Metamorphic operations Relaxed states are generated by removing a random car ahead of the agent, thus reducing the chance of crashing (see Figure 2.2a). Conversely, unrelaxed states are generated by adding a car at a safe distance from the agent (not leading to unavoidable crashes).

`DIV_THRESH` **and** `WALK_LENGTH` We set `DIV_THRESH` to 3.6 to capture the typical scale of diversity in this domain, which we gauged by inspecting the visual differences between states. We set `WALK_LENGTH` to 3 to balance the typical risk of random walks, which quickly lead to crashes in this domain.

2.8.2 LunarLander

The LunarLander domain consists of an uneven lunar surface and a lander with two legs. The lander appears at the top of the environment with a random velocity vector, and the task is to land it on its legs – if the body touches the surface, the lander crashes. There are four discrete actions: firing the bottom engine, the left-hand side engine, the right-hand side engine, noop. The effect of firing an engine is stochastic, following a probability distribution over the yielded force. Touching a leg to the ground yields reward $+100$,



(a)

(b)

(c)

Figure 2.2: Illustrations of domains and relaxations (as per Definition 2) used in our evaluation: (a) Highway, (b) LunarLander, and (c) BipedalWalker.

and touching the body to the ground yields reward -100. There is no discount factor, and the best-case reward is over 200.

Policy training We train the agent using PPO Schulman *et al.* (2017) implemented in the SB3 library Raffin *et al.* (2019). Our agent is well trained, and achieves an average reward of 205 points (after 1 million training episodes).

Metamorphic operations Relaxed states are generated by decreasing the height of the surface, giving the lander more time to land (see Figure 2.2b). Conversely, we generate unrelaxed states by increasing the surface height up to a safe distance.

`DIV_THRESH` **and** `WALK_LENGTH` We set `DIV_THRESH` to 0.65 and `WALK_LENGTH` to 25.

2.8.3 BipedalWalker

In the BipedalWalker domain, a bipedal robot moves along a finite-length terrain that has a rough surface. The robot's task is to move forward until the end of the terrain. The action space is continuous, with actions being defined by a 4-tuple of numbers $x_i \in [0.0, 1.0]$. Each x_i specifies the force applied to one of the joints of the robot. The actions are deterministic; the only stochastic elements in this domain are the terrain (surface) shape and the initial forces in the robot's joints. The best-case achievable reward is $+300$ collected when reaching the end of the terrain, plus small positive rewards that can be collected beforehand. If the robot falls, it receives -100 points, and the game ends immediately. Again, there is no discount factor.

Policy training We use the PPO algorithm from SB3 for training. Our agent achieves an average reward of 302 points (after 1 million training episodes).

Metamorphic operations As smooth surfaces are easier to navigate for the walker, relaxed states are generated by making the terrain smoother (see Figure 2.2c), whereas unrelaxed states are generated by making the terrain rougher.

`DIV_THRESH` **and** `WALK_LENGTH` We set `DIV_THRESH` to 2.0 and `WALK_LENGTH` to 25.

Overall, our case studies explore moving obstacles under simple action dynamics (Highway), fixed obstacles under complex action dynamics (LunarLander), and keeping balance with continuous motor control (BipedalWalker). They, thus, cover a broad range of applications featuring obstacle avoidance, which is a ubiquitous problem in neurally controlled systems.

2.9 Experiments

Our primary evaluation concerns bug-finding capability, i.e., the number of (seed-)bugs correctly identified by different oracles. We, furthermore, analyze the impact of the `POL_PROB` and `DIV_THRESH` algorithm parameters, and we provide data on runtime to gauge the practical effort needed. In what follows, we first introduce the oracles we

compare, then focus on these three evaluations in turn.

To account for the randomness in our algorithms and game environments, we run each experiment 8 times and report statistics over these runs below. Each run was performed on a Debian 10 machine with 1.5 TB of memory and 96 CPUs (Intel(R) Xeon(R) Gold 6248R CPU @ 3.00GHz). For each run, we used a time limit of 24 hours.

2.9.1 Competing Oracles

To provide a comprehensive evaluation, we consider not only our metamorphic oracles, but eight oracles in total:

MMBug, MMSeedBugBasic, MMSeedBugExt These are the oracles from Algorithm 1. (MM stands for metamorphic.)

FailureSeedBug This oracle just flags s_i as a bug if $E^r.\sigma[\pi, s_i]$ fails.

This is a trivial baseline that does not check avoidability.

RuleSeedBug This oracle uses the aforementioned "change nothing" rule (see Section 2.2), reporting pool state s_i as a seed-bug if there is an unrelaxed state t_i, $R(t_i, s_i)$, such that $E^r.\sigma[\pi, t_i]$ succeeds and $\pi(s_i) \neq \pi(t_i)$. The rationale is that what works for t_i also works for s_i, so the policy should not change.

As $\pi(s_i) = \pi(t_i)$ is possible but not necessary, this oracle may incorrectly classify s_i as a seed-bug. Here, we report only the true positives, measuring the oracle's ability to identify true bugs (which as we shall see is lacking).

PerfectBug and PerfectSeedBug These oracles provide exact measuring lines. PerfectSeedBug explores all possible trajectories from a given state with the given random seed, and detects a bug only if there exists a winning trajectory but the policy fails from this state. PerfectBug simply measures $P_\phi(\pi, s)$ and $P^*_\phi(s)$, and detects a bug if $P_\phi(\pi, s) > P^*_\phi(s)$. Computing these oracles is tractable only for Highway, so we report data only for this case study.

MMSeedBug2Bug This oracle first calls MMSeedBugBasic. If MMSeedBugBasic flags s_i as a bug due to unrelaxed state t_i, then MMSeedBug2Bug flags s_i as a bug if additionally $P_\phi(\pi, t_i) < P_\phi(\pi, s_i)$.

Such seed-bug filtering speeds up bug-finding as we will see. Further, this oracle evaluates how many seed-bugs found by MMSeedBugBasic correspond to bugs.

In MMBug and MMSeedBug2Bug, we evaluate P_ϕ by running the policy 30 times. Based on limited experiments, this is reasonable in our case studies; using statistical methods to compute P_ϕ up to a confidence bound is future work.

Prior to considering the empirical data for these oracles, note the following guaranteed relations between the sets of states (or state/seed pairs) they identify as bugs:

RuleSeedBug ⊆ MMSeedBugBasic The true positives of the RuleSeedBug oracle are dominated by those of MMSeedBugBasic, because if $R(t_i, s_i)$ and $E^r.\sigma[\pi, t_i]$ succeeds, then s_i is a bug iff $E^r.\sigma[\pi, s_i]$ fails – which is precisely what MMSeedBugBasic is checking.

MMSeedBugBasic ⊆ PerfectSeedBug, MMBug ⊆ PerfectBug By Proposition 3.

PerfectSeedBug ⊆ FailureSeedBug FailureSeedBug catches all seed-bugs but may incorrectly flag non-bugs.

MMSeedBug2Bug ⊆ MMSeedBugBasic, MMSeedBug2Bug ⊆ MMBug By construction.

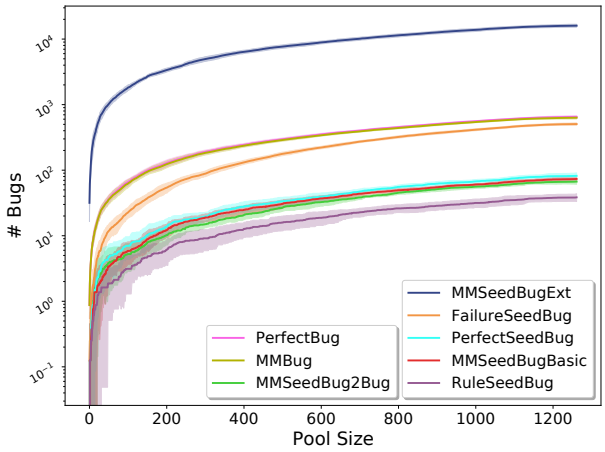
MMSeedBug2Bug = MMSeedBugBasic = MMBug This relation holds on deterministic domains where policy runs are unique.

The MMSeedBugExt oracle is incomparable to the others as it is the only one that attempts to find additional (seed-)bugs, beyond the pool states s_i.

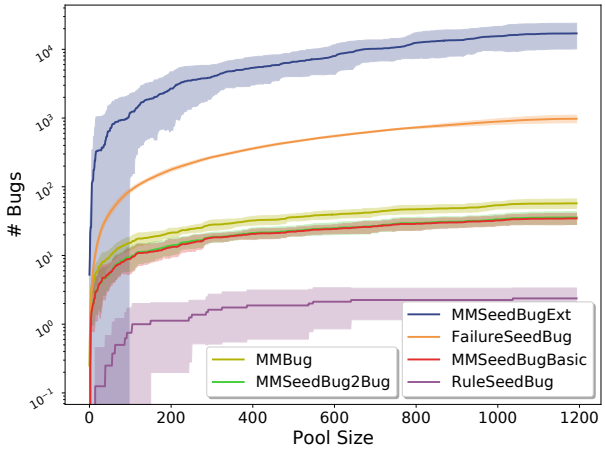
2.9.2 Results: Oracle Capability

Figure 2.3 shows our evaluation of oracle bug-finding capability. We fix the default version of the fuzzer here (using the parameter settings as previously specified). For each case study, we plot pool size on the x-axis as the testing progresses, and we show how many bugs were reported by each oracle on the y-axis; dark lines denote mean values and shaded areas standard deviation.

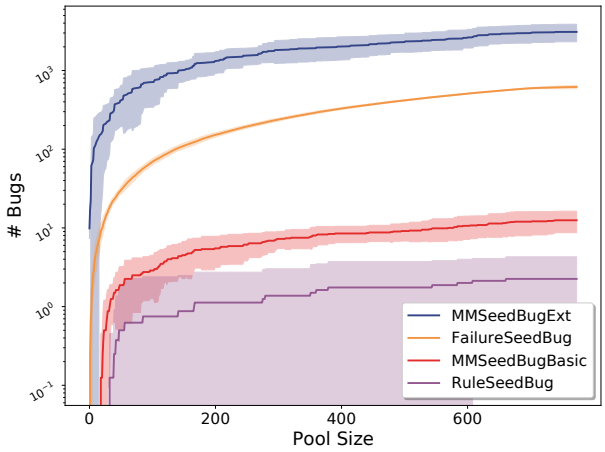
Consider first Figure 2.3a about Highway, where exact measuring lines by perfect oracles are available. These measuring lines attest to the strength of our metamorphic oracles in this domain: MMBug is close to PerfectBug, and MMSeedBugBasic is close to PerfectSeedBug. The average numbers of (seed-)bugs identified at the end of testing are 631.6 for MMBug, 650.1 for PerfectBug, 73.5 for MMSeedBugBasic, and 82.0 for PerfectSeedBug. Among the seed-bug oracles, FailureSeedBug reports many false positives, RuleSeedBug lags behind MMSeedBugBasic (38.5 at the end), and MMSeedBugExt finds a large number of additional bugs ($\sim$ 16000). The gap between the seed-bug and bug oracles is large here, because the former ignore pool states that are solved by the policy under the one random seed chosen by the fuzzer, whereas the latter consider multiple seeds and identify many bugs for the same pool states. Accordingly, while MMSeedBug2Bug is close to MMSeedBugBasic showing that most seed-bugs we identify are bugs, MMSeedBug2Bug lags far behind MMBug.



(a) Highway



(b) LunarLander



(c) BipedalWalker

Figure 2.3: Evaluation of oracles: number of (unique) bugs as a function of pool size during the testing process. MMSeedBug2Bug and MMBug are not included in BipedalWalker as it is a deterministic environment, so these oracles coincide with MMSeedBugBasic.

Consider now LunarLander and BipedalWalker in Figure 2.3b and 2.3c. MMSeedBug-Basic vastly outperforms RuleSeedBug. FailureSeedBug is far above that, but *at least* 50% of these failures are unavoidable. We calculate this number by implementing a limited-budget depth-first search, which examines all possible trajectories from a given state within a 15-minute time budget. We run this search on the pool states for which the policy fails, and we report the states for which a crash is unavoidable. There is such a large proportion of states where a crash is unavoidable because the LunarLander agent can easily get to unrecoverable states with random actions. MMSeedBugExt finds many additional bugs as before ($\sim$ 17000).

In LunarLander, MMBug is only slightly above MMSeedBugBasic (in difference to Highway); MMSeedBug2Bug and MMSeedBugBasic are so close to each other that the two plots cannot be distinguished (99% of the seed-bugs reported by MMSeedBugBasic are bugs here). In BipedalWalker, the three plots necessarily coincide as the environment is deterministic. For this reason, we do not plot these lines.

Overall, the results show that **metamorphic oracles are highly useful for identifying bugs in action policies**, and they are superior to the rule-based and failure-based alternatives we evaluate. In the two domains where we are able to check (one with limited budget), the oracles are close to perfect. Moreover, **seed-bug detection is a practical proxy for bug detection** in the sense that most seed-bugs detected by MMSeedBugBasic are bugs. Especially, **MMSeedBugExt is extremely effective in identifying bugs in diverse states**.

2.9.3 Results: Fuzzer Configurations

For our evaluation of fuzzer configurations – specifically, algorithm parameters `POL_PROB` and `DIV_THRESH`, which are the most interesting as discussed – see Table 2.1.

First, consider the impact of `POL_PROB`, controlling whether or not the policy under test is used to (partially) inform the random walks in the fuzzer. This is intended to improve the bug-finding capability in domains where purely random walks incur too many unavoidable failures. This effect is observed across domains, but it is especially noticeable in LunarLander, where a non-zero vs. a zero `POL_PROB` results in finding significantly more bugs for each setting of `DIV_THRESH`. (Note that value 0> in the table was 0.029 for `POL_PROB=0.2` and 0.026 for `POL_PROB=0` in our experiments.) A clear added value of finding more bugs is in using them as part of a targeted re-training process; as the amount of training data is important, so is the number of bugs.

On the other hand, a non-zero diversity threshold for adding states to the pool (`DIV_THRESH`) reduces the number of bugs found in LunarLander by up to an order of magnitude, with smaller reductions in Highway and BipedalWalker. This is because some detected bugs

are not added to the pool, and there is a computational overhead associated with checking for diversity. The desired effect of increasing bug diversity is achieved though; all minimum, maximum, and average distance values among bug states are higher for a non-zero vs. a zero threshold. In general, diversity is important in gaining confidence that a policy is correct. Moreover, more diverse bugs could be more effective for re-training as they may point out different policy shortcomings. However, the gain in diversity requires more runtime, as we also show next.

Table 2.1: Evaluation of fuzzer configurations: number and diversity of bugs at the end of the testing process, using the MMSeedBugBasic oracle.

SETTING / DOMAIN	DIV_THRESH>0							
	POL_PROB=0.2				POL_PROB=0			
	# Bugs	**Distance** (L_2)			**# Bugs**	**Distance** (L_2)		
black!25	black!25	Min	Max	Avg	black!25	Min	Max	Avg
Highway	73.5	3.6	59.8	12.1	50.5	3.6	62.6	13.6
LunarLander	34.6	0.7	3.5	1.6	15.3	0.7	3.1	1.4
BipedalWalker	13.3	2.0	6.3	3.8	13.0	2.0	6.3	3.6

SETTING / DOMAIN	DIV_THRESH=0							
	POL_PROB=0.2				POL_PROB=0			
	# Bugs	**Distance** (L_2)			**# Bugs**	**Distance** (L_2)		
black!25	black!25	Min	Max	Avg	black!25	Min	Max	Avg
Highway	116.5	1.1	42.3	10.5	90.7	1.1	37.9	9.7
LunarLander	261.0	0>	2.7	1.0	167.2	0>	2.7	1.0
BipedalWalker	14.5	1.0	4.2	2.7	13.7	1.4	4.2	2.4

Table 2.2: Oracle runtime (in seconds) per state for MMSeedBugBasic (MMSBB), MMBug (MMB), and MMSeedBug2Bug (MMSB2B).

DOMAIN	ORACLE		
black!25	**MMSBB**	**MMB**	**MMSB2B**
Highway	0.4	10.1	0.6
LunarLander	0.6	10.1	1.4
BipedalWalker	6.1	n/a	n/a

Table 2.3: Fuzzer runtime required to add one more state to the pool.

DOMAIN	DIV_THRESH>0		DIV_THRESH=0	
black!25	POL_PROB=0.2	POL_PROB=0	POL_PROB=0.2	POL_PROB=0
Highway	73.0	96.9	0.001	0.003
LunarLander	82.4	158.0	0.004	0.004
BipedalWalker	119.6	159.0	0.008	0.010

2.9.4 Results: Fuzzer and Oracle Runtime

Finally, consider the runtime data in Table 2.2 and Table 2.3. We evaluate the MMSeedBugBasic and MMBug oracles to assess the effort required for identifying seed-bugs vs. bugs. We also include MMSeedBug2Bug to assess the speed-up gained from using seed-bug detection as a filter. As expected, seed-bug detection is much faster than bug detection, and seed-bug filtering gets rid of much of the overhead (at the risk of missing bugs, cf. above). Note that we do not evaluate MMBug and MMSeedBug2Bug for BipedalWalker as it is deterministic; these oracles coincide with MMSeedBugBasic.

Regarding fuzzer parameters, with DIV_THRESH>0, finding a new state for the pool takes 4–5 orders of magnitude more time (on average; at the start of testing, the overhead is smaller). Given that its only advantage is bug diversity (cf. Table 2.1), whether or not that switch should be activated depends on the application.

2.9.5 Feasability Study

The π-fuzz framework proves to be very effective in detecting avoidable, bad policy behavior in 2D-world, single-agent, obstacle-avoidance games with fixed or moving obstacles. In fact, for games where it was tractable to compute an optimal policy, π-fuzz was almost as effective in detecting avoidable policy failures as using the optimal policy. Other oracles either missed many policy bugs or reported many spurious ones.

As a natural nextstep, we explore whether the policy bugs found by π-fuzz are actionable. While we leave a comprehensive exploration as a future work, we performed a feasibility study to investigate whether the reported violations are actionable. Specifically, we investigate how to incorporate detected bug states during training to obtain better-behaved action policies, similar to how adversarial training is used to obtain more robust models Madry *et al.* (2018).

On a high level, we adjust existing policy training algorithms to start runs not only from random initial states, but also from bug states. Our guided-training algorithm *automatically* determines the probability with which bug states should be incorporated.

To obtain bug states, our algorithm runs π-fuzz to test the policy at regular, adjustable intervals during training, that is, it alternates between training and testing phases. All detected bug states are added to a list of bug states, and guided training uses this list to select states to incorporate in subsequent training phases. We investigate the effect of different bug selection strategies, e.g., prioritizing more recently detected bug states, but we always retain bug states detected in all previous testing phases to prevent the policy from "forgetting" about them.

Bug Oracle For Training. The π-fuzz framework already introduces metamorphic oracles for detecting policy bugs. However, these oracles are impractical for the specific use case of training better policies. In particular, they either depend on a fixed random seed (*i.e.* MMSeedBugBasic, MMSeedBugExt), thereby behaving irregularly across different seeds, or they are computationally expensive (*i.e.* MMBug). As a result, we define a metamorphic policy-bug oracle for training, called BUGORACLEFORTRAINING and shown in Algorithm 3. Simply, if the policy performs better for harder s than for easier t for b consecutive times, the oracle returns that a bug is found.

Algorithm 3: Policy-bug oracle for training

```
1 Function BUGORACLEFORTRAINING(π, E, b, s, t):
2     while 0 < b do
3         r := GETRANDOMSEED();
4         if E^r.σ[π, s] ≤ E^r.σ[π, t] then
5             return 0;
6         b := b − 1;
7     return 1;
```

Guided-Training Algorithm. Using this oracle, we now design a guided-training algorithm, Algorithm 4, that aims to obtain policies exhibiting fewer bugs without sacrificing their performance (in terms of reward). More specifically, we adapt canonical policy training. We highlight these adaptations with comments in Algorithm 4.

The algorithm takes the following inputs: an initial policy π, an environment E, a corpus C of pairs of states (s,t), where t relaxes s, the test frequency f of the policy, and a scaling coefficient α, which scales the probability of incorporating bug states during training.

On line 2, the algorithm initializes a list of bug states, on line 4, it initializes the probability of guiding training with bug states, and on line 6, it initializes a dictionary of average rewards collected during training. The loop on line 8 iterates over the total number of

Algorithm 4: Guided training

```
1  Function GuidedTraining(π, E, C, f, α):
       /* initialize list of bug states                          */
2      S_B := [];
3      /* initialize probability of guiding training             */
       /* with bug states                                        */
4      p_B := 0;
5      /* initialize dictionary of average rewards               */
6      rwrds := EmptyDict();
7      /* iterate over total number of batches                   */
8      for b := 0 to batches do
           /* initialize list of rollouts in current batch       */
9          rollouts := [];
           /* iterate over batch size                            */
10         for i := 0 to batchSize do
11             if WithProb(p_B) ∧ S_B ≠ [] then
                   /* select bug state                           */
12                 s := SelectBugState(S_B);
13             else
14                 /* select random initial state                */
15                 s := E.randomInit();
               /* run policy and store rollout                   */
16             rollouts := rollouts ++ [E*.σ[π, s]];
           /* train policy                                       */
17         π := TrainPolicy(π, rollouts);
18         if ShouldCalcProb(f, b, batches) then
               /* calculate average policy reward                */
               /* on non-bug states                              */
19             rwrds[b] := GetAvgRwrd(π, E);
               /* adjust p_B based on policy rewards             */
20             p_B := CalcProb(rwrds, α);
21         if ShouldTest(f, b, batches) then
               /* test policy to detect new bug states           */
22             S'_B := TestPolicy(π, E, C);
               /* update list of bug states                      */
23             S_B := S_B ++ S'_B;
24     return;
```

batches, that is, $timesteps/batchSize$. As in canonical policy training, each loop iteration initializes a list of rollouts collected during the current batch (line 9).

The loop on line 10 runs the policy as many times as the batch size. Each iteration selects a state from which to start the policy run (lines 11–15), runs the policy from the selected state, and stores the rollout (line 16). A state is selected as follows: with probability p_B and if S_B is non-empty (line 11), line 12 selects a bug state; otherwise, a random initial state is chosen (line 15) as usual. Note that $E.reset()$ returns a random initial state in S_I.

On line 17, the algorithm trains the policy over the list of rollouts. Lines 18–20 update probability p_B based on the policy performance, measured in achieved rewards. Function SHOULDCALCPROB (line 18) uses the provided frequency f to determine whether the probability should be updated. In our implementation, the probability updates happen with a frequency of $5f$; they do not happen in every loop iteration to keep the runtime overhead low. In particular, line 19 calculates the average reward that the current policy achieves on a set of (non-bug) random initial states; this, of course, involves running the policy on each of these states. We use non-bug states because we aim to achieve a policy performance that is comparable to normal training (without any guidance). On line 20, p_B is updated based on the collected rewards and α, a scaling coefficient. We define function CALCPROB later in this section.

Lines 21–23 test the current policy and enrich the list of bug states. On line 21, function SHOULDTEST allows testing π with frequency f. Again, the policy is not tested in every loop iteration to avoid a prohibitively large overhead; we explore the effect of different values for f in our experimental evaluation. The policy is tested by function TESTPOLICY on line 22, which takes as input the corpus of pairs of states C and invokes oracle BUGORACLEFORTRAINING (Algorithm 3) on pairs of states (s,t). It returns a list S'_B of detected bug states, which is added to S_B on line 23. Note that the corpus C is generated by π-fuzz Eniser *et al.* (2022). First, π-fuzz generates a set of diverse states s by taking random or informed actions (i.e., predicted by a well trained policy) from an arbitrary initial state. These states are then filtered by L_2 distance to ensure diversity—state diversity enables more effective testing of the policy. Next, C is formed by relaxing each state s to obtain t, and therefore, a pair (s,t).

In the following, we discuss in more detail the functions that are called in Algorithm 4 but have not yet been defined.

Bug-state selection. The SELECTBUGSTATE function selects a bug state from list S_B (line 12 of Algorithm 4), which contains all bug states detected so far, from least to most recent. The function may use one of two distributions to perform the selection, a uniform and an exponential distribution, which prioritizes more recent bug states. In both cases, we retain and may use all detected bug states during training to prevent the policy from

Algorithm 5: Calculating the probability of incorporating bug states during training

```
1 Function CalcProb(rwrds, α):
2     rwrds_F := ApplySavGolFilter(rwrds);
3     rwrds_N := NormalizeRewards(rwrds_F);
4     slope := |ComputeAvgSlope(rwrds_N)|;
5     return α * (1 − slope);
```

"forgetting" about them.

Probability calculation. Probability p_B determines how often a bug state is preferred over a random initial state during training (lines 11–15 of Algorithm 4). Striking a good balance between training on bug states versus random initial states is challenging. In particular, using too many bug states might hinder policy learning and lead to poor performance. On the other hand, using too few bug states might be insufficient for training a better behaved policy.

Function CalcProb, called on line 20 of Algorithm 4, calculates probability p_B of incorporating bug states during training. On a high level, the more stable the current policy performance, the higher the probability of guiding training with bug states, and vice versa. The policy reward or loss may be used as a performance indicator—we use achieved rewards in Algorithm 4 and our implementation.

Algorithm 5 defines function CalcProb. Since rewards may fluctuate considerably during training, we first smoothen them by applying the Savitzky-Golay filter Savitzky and Golay (1964) (line 2). Next, we normalize the resulting rewards in the range $[0,1]$ (line 3) and compute the current slope of the reward line (over the number of batches). The absolute value of this slope is stored in variable *slope* on line 4 and is also in the range $[0,1]$. On line 5, the function computes the probability of guiding training with bug states as follows. The expression $(1 - slope)$ evaluates to a lower probability for steeper changes in policy performance, and a higher probability for more stable changes. This probability is then scaled by coefficient $\alpha \in [0,1]$ and returned. We introduce α to provide users with a degree of control over the value of p_B and experiment with different values of α in our evaluation.

Comparison. In this feasability study we investigate finding policies that are less buggy while maintaining the same level of reward as normal training. Therefore, in a guided training campaign, we seek for policies whose reward is as high as the maximum reward that is found in a normal training campaign. We then compare the number of bugs in those policies. However, depending on the use case, one might prioritize reducing bugs

by sacrificing from reward. Therefore, we also compare the number of bugs in guided and normal policies that achieve a slightly lower reward than the maximum achievable with normal training.

We implement this idea by testing and evaluating a policy during its training with fixed intervals and storing it along with the found bug and reward. Upon completion of a normal training campaign, we identify the policy with the highest reward and others with slightly lower rewards (e.g. 1%, 3% and 5% less). We then report the minimum number of bugs associated with these policies, considering the maximum allowable sacrifice in reward. To compare guided training performance, we use the maximum and sacrificed reward values from normal training as reference points and report the minimum number of bugs accordingly. To account for randomness, we repeat each training run with six differnt random seeds. If a guided training run for a specific seed fails to achieve the reference reward from normal training, we discard that run. This ensures our comparisons focus only on policies that perform as well as those from normal training in terms of reward. To present results, we draw boxplots of bugs from no sacrifice to at most 5% sacrifice from the maximum reward achieved with normal training.

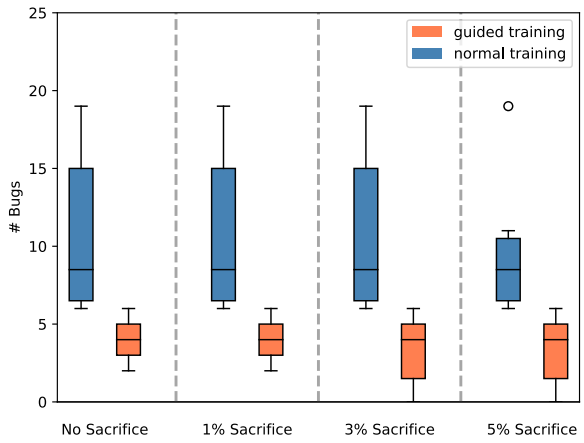
Figure 2.4 shows the result of our comparison for the default configuration of guided training. We obtain these plots with experiments where we set α to 0.5, test frequency to 0.02 and we conduct bug selection with exponential distirbution. Each box shows the minimum number of bugs for the policies across 6 runs, the black line is a median of these 6 values, and the error margin shows all values of bugs for policies with the corresponding reward value (max, up to 99%, up to 97%, and up to 95%). Blue boxes denote policies obtained with normal training, and orange - with guided training.

The results show that incorporating bug states can help reduce policy bugs to some extent. In the Highway domain, we observe fewer bugs with no sacrifice in reward. For LunarLander, the benefits of this technique become evident only after a 3% sacrifice in reward. However, in BipedalWalker, we do not observe any benefit from guided training. We conclude that there is still room for improvement in this direction. Specifically, it is challenging to find a single parameter setting that works well across all benchmarks. Our further experiments indicated that understanding the effects of parameters such as *alpha*, test frequency, and bug state selection requires more investigation.

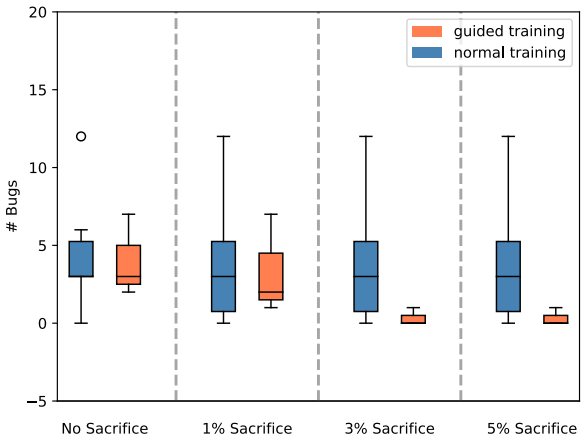
2.9.6 Threats to Validity

We identify the following threats to the validity of our experiments.

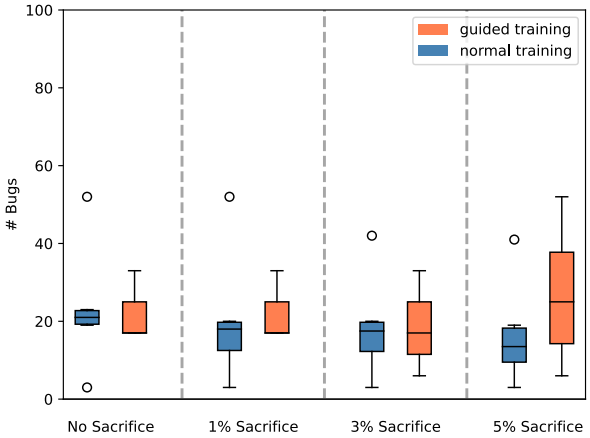
Games. The detected policy bugs, of course, depend on our selection of games. However, apart from the Highway benchmark that we developed, we show the effectiveness



(a) Highway



(b) Lunar lander



(c) BipedalWalker

Figure 2.4

and generality of our approach by additionally applying it to two popular, off-the-shelf Gym environments. Further, we consider both deterministic and stochastic games.

Agents. The detected bugs also depend on the quality of the agents we train. However, all our agents are well trained as we describe in Section 2.8.

Metamorphic operations. The effectiveness of our approach is affected by the metamorphic operations that we define (see Section 2.8). However, as described earlier, relaxing obstacles is easy, hardly requiring any domain knowledge. When adding obstacles, we do so carefully to prevent causing unavoidable crashes, *e.g.*, by adding a car at a safe distance from the agent in Highway, or by increasing the surface height up to a safe distance in LunarLander.

2.10 Conclusion

Testing action policies for avoidable failures requires oracles that can effectively identify sub-optimal failure-avoiding abilities. We have shown that such oracles can be obtained from relaxations, by adapting ideas from metamorphic testing. Our experiments confirm the potential of this approach.

This work opens up an entire universe of exciting research on relaxation-based metamorphic oracles. Possibilities include the automated design of state relaxations and thus metamorphic oracles; intelligent methods to explore environment behaviors in a search for seed-bugs; fault localization trying to identify specific combinations of policy decisions leading to failures; as well as closing the loop with re-training by feeding bug states back into reinforcement learning, until testing yields sufficient confidence in the policy.

Chapter 3

Specifying and Testing Safety Properties of ML Models with NOMOS

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 chapter, we present our work where 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.

3.1 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); Zeng *et al.* (2023); Khedr and Shoukry (2023); Banerjee *et al.* (2024)).

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 chapter:

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. Recall LunarLander from the previous chapter, the popular Gym Brockman *et al.* (2016); Towers *et al.* (2024) environment consisting of a 2D-world with an uneven lunar surface and a reinforcement-learning (RL) lander agent. An expected hyperproperty for this agent 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". Also recall BipedalWalker, the other Gym environment where a bipedal robot moves along a finite-length terrain that has a rough surface. A straightforward hyperproperty would be that "if the walker successfully reaches the end of the terrain, then making the terrain smoother should also result in successfully reaching the end." In both games, 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 chapter, 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. 3.3) 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 chapter 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.

3.2 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. 3.1a expressing the COMPAS property described earlier. On line 1, we specify that we need an input `x1`, *i.e.*, a criminal. Lines 2–4 get the first feature of `x1`, which corresponds to the number of felonies, and assign it to variable `v1`; in variable `v2`, we increase this number, and create a new criminal `x2` that differs from `x1` only with respect to this feature, *i.e.*, `x2` has committed more felonies than `x1`. Line 5 specifies a precondition that the new criminal's felonies should not exceed a sensible limit. Lines 6–7 declare two outputs, `d1` and `d2`, that are assigned the model's prediction when calling it with criminal `x1` and `x2`, respectively

[1] https://github.com/Rigorous-Software-Engineering/nomos

```
input x1;
var v1 := getFeat(x1, 1);
var v2 := v1 + randInt(1, 10);
var x2 := setFeat(x1, 1, v2);
requires v2 <= 20;
output d1;
output d2;
{
  d1 = predict(x1)
  d2 = predict(x2)
}   # 0-low, 1-medium, 2-high
risk ensures d1 <= d2;
```

(a) COMPAS 2-safety property.

```
input x1;
var x2 := blur(x1);
ar v1 := label(x1);
output d1;
output d2;
{
  d1 = predict(x1)
  d2 = predict(x2)
}
ensures d2==v1 ==> d1==v1;
```

(b) MNIST 2-safety property.

```
input x1;
input x2;
var v1 := getFeat(x1, 1);
var v2 := getFeat(x2, 1);
var v3 := strConcat(v1, v2);
var x3 := setFeat(x1, 1, v3);
output d1;
output d3;
{
  d1 = predict(x1)
  d3 = predict(x3)
} # 0-pos, 1-neg
ensures d1 <= d3;
```

(c) HotelReview 2-safety property.

```
input s1;
var s2 := relax(s1);
output o1;
output o2;
{
  o1, o2 = 0, 0
  for _ in range(10):
    rs = randInt(0, MAX_INT)
    o1 += play(s1, rs)
    o2 += play(s2, rs)
}   # 0-lose, 1-win
ensures o1 <= o2;
```

(d) LunarLander 20-safety property.

Figure 3.1: Example k-safety specifications in NOMOS.

```
<spec>         ::= { <import> } <input> { <input> }
                   { <var_decl> } { <precond> } { <output> }
                   "{" <code> "}" { <postcond> }
<import>       ::= "import" <model_name> ";"
<input>        ::= "input" <var_name> ";"
<var_decl>     ::= "var" <var_name> ":=" <scalar_expr> ";" |
                   "var" <var_name> ":=" <record_expr> ";"
<precond>      ::= "requires" <bool_expr> ";"
<output>       ::= "output" <var_name> ";"
<postcond>     ::= "ensures" <bool_expr> ";"
<scalar_expr>  ::= <bool_expr> |
                   <int_expr> | <string_expr>
                   "getFeat(" <record_expr> "," <int_expr> ")" |
                   "label(" <record_expr> ")" |
                   "randInt(" <int_expr> "," <int_expr> ")" |
                   "strConcat(" <string_expr> "," <string_expr> ")"
<bool_expr>    ::= <bool_literal> |
                   <var_name> | "!"<bool_expr> |
                   <bool_expr> "&&" <bool_expr> |
                   <scalar_expr> "==" <scalar_expr> |
                   <scalar_expr> "<" <scalar_expr> |
                   <record_expr> "==" <record_expr>
<record_expr>  ::= <var_name> |
                   "setFeat(" <record_expr> "," <int_expr> "," <scalar_expr> ")"|
                   "blur(" <record_expr> ")" |
                   "wNoise(" <record_expr> ")" |
                   "relax(" <record_expr> ")" |
                   "unrelax(" <record_expr> ")"
```

Figure 3.2: The NOMOS grammar.

(see block of Python code on lines 8–11). Finally, on line 12, we specify the postcondition that the recidivism risk of criminal `x2` should not be lower than that of `x1`.

Fig. 3.1b shows the MNIST specification. Given an image `x1` (line 1), image `x2` is its blurred version (line 2), and variable `v1` 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. 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 `blur`, we used function `wNoise` adding white noise to audio.

The HotelReview specification is shown in Fig. 3.1c. 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, `x1` and `x2`, and in variables `v1` and `v2` on lines 3–4, we store their negative sections (feature `1` retrieved with function `getFeat`). We then create a third review, `x3`, which is the same as `x1` except that its negative section is the concatenation of `v1` and `v2` (lines 5–6). The postcondition on line 13 checks that the detected sentiment is not more positive for review `x3` than for `x1`.

Finally, consider the LunarLander specification in Fig. 3.1d. On line 1, we obtain an

input `s1`, which is an initial state of the game. Line 2 "relaxes" this state to obtain a new state `s2`, which differs from `s1` 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 `o1` and `o2` to zero and play the game from each initial state, `s1` and `s2`, in a loop; `o1` and `o2` 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 `s2` is indeed easier, *i.e.*, that stochasticity cannot make it harder, by using the same seed on lines 9–10. Note that function `play` invokes the policy multiple times (*i.e.*, after every step in the game simulator). Finally, line 12 ensures that, when playing the easier game (starting with `s2`), the number of wins should not decrease. Since this property depends on 20 model invocations, it is a 20-safety property[2]. Conversely, we can also make the game harder by "unrelaxing" the original initial state, *i.e.*, increasing the surface height.

Grammar. Fig. 3.2 provides a formal grammar for NOMOS (in a variant of extended Backus-Naur form). The top-level construct is `<spec>` on lines 1–3. It consists of zero or more `import` statements—the curly braces denote repetition—to import source-code files containing custom implementations for domain-specific functions, *e.g.*, `blur` or `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 5–10). For instance, a precondition (line 8 of Fig. 3.2) consists of the token `requires`, a Boolean expression, and a semicolon. For brevity, we omit a definition of `<code>`; 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 `<model_name>` and `<var_name>`.

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 11–16), Boolean (lines 17–22), and record expressions (lines 23–28). 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.*, `getFeat` and `setFeat` retrieve and modify record fields. Integer and string expressions are defined as expected.

[2]This property could be reformulated as a 2-safety property by abstracting the system to return the outcome of all 10 plays in a single call.

3.3 Testing Framework for NOMOS

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 Sun *et al.* (2022); He *et al.* (2020), NLP models Ma *et al.* (2020), object-detection systems Wang and Su (2020), action policies Eniser *et al.* (2022); Mazouni *et al.* (2025), and autonomous cars Tian *et al.* (2018); Zhang *et al.* (2018); Luu *et al.* (2024).

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. 3.1a, lines 2–5 describe the transformation to input `x1` in order to obtain `x2`, and line 12 specifies the relation between the corresponding outputs. We, therefore, design the framework in Fig. 3.3 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. 3.1a, a bug would comprise two concrete instances of a criminal, c_1 and c_2, such that (1) c_2 differs from c_1 only in having more felonies, and (2) the recidivism risk of c_2 is predicted to be lower than that of c_1.

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. 3.1a is shown in Fig. 3.4. 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. 3.4, 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 2-10). Specifically, for every `input` declaration, the

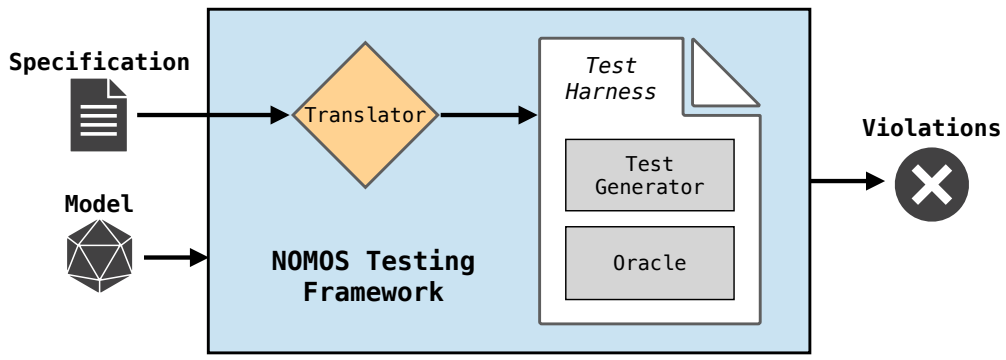


Figure 3.3: An overview of our testing framework.

test generator randomly selects an input from a source specified in the imported files (line 3)—note that `import` statements are not shown here but are defined on line 4 of Fig. 3.2. 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 4–6). 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 8–10).

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 de-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. 3.4, there is randomness on lines 3 and 5). 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, *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.

3.4 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

```
while budget > 0:
  # test generator
  x1 = compas.randInput()
  v1 = compas.getFeat(x1,1)
  v2 = v1 + compas.randInt(1,10)
  x2 = compas.setFeat(x1,1,v2)

  if not(v2 <= 20):
    compas.prec_viol += 1
    continue

  # code
  d1 = compas.predict(x1)
  d2 = compas.predict(x2)

  # oracle
  if d1 <= d2:
    compas.passed += 1
  else:
    compas.postc_viol += 1
    compas.process_bug()

  budget -= 1
```

Figure 3.4: Snippet of generated harness for the specification of Fig. 3.1a.

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:

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 $max\_depth = 8$. For GermanCredit, we trained an NN with 1 hidden layer of size 10 and ca. 1400 parameters as well as a DT with $max\_depth = 6$. For COMPAS, 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, *i.e.*, a perfectly accurate model could violate the property, whereas a less accurate model might not.

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.

Speech. We pre-processed the SpeechCommand dataset Warden (2018) to convert wave-

forms 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.

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.

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.

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. 3.4), 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 (*e.g.*, for the number of bugs) unless stated otherwise. In all harnesses except for LunarLander and BipedalWalker, the input source (*e.g.*, line 3 of Fig. 3.4) is the test set. For LunarLander and BipedalWalker, the input source is a pool of game states that was generated by 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, *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 to 13

Table 3.1: Number of specified properties, violated properties, and unique bugs per dataset and model.

Dataset	Model	Properties		Unique
		Specified	Violated	Bugs
COMPAS	NN	12	7	960.0
	DT	12	6	294.8
GermanCredit	NN	10	6	295.2
	DT	10	6	286.9
MNIST	NN	1	1	15.6
SpeechCommand	NN	1	1	14.2
HotelReview	NN	4	4	3288.0
LunarLander	NN	2	2	3459.0
BipedalWalker	NN	2	2	443.5

hours for BipedalWalker. The total amount of compute for all experiments is ca. 1.5 day on the above machine.

Results. Tab. 3.1 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. 3.2. 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. 3.2). 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.

Table 3.2: Average number of unique bugs for each specification.

Dataset	Specification	Unique Bugs	
		NN	DT
COMPAS	Felony Inc	42.9	3.5
	Felony Dec	0.0	0.0
	Misdmnr Inc	619.7	0.0
	Misdmnr Dec	4.0	0.0
	Priors Inc	0.5	90.0
	Priors Dec	0.0	97.2
	Others Inc	289.0	91.1
	Others Dec	3.0	8.0
	IsRecid Set	0.0	0.0
	IsRecid Unset	0.0	0.0
	IsVRecid Set	0.9	5.0
	IsVRecid Unset	0.0	0.0
GermanCredit	Crdt Amount Inc	0.0	122.3
	Crdt Amount Dec	0.0	75.4
	Crdt Hist Inc	78.1	8.0
	Crdt Hist Dec	122.8	31.2
	Empl Since Inc	13.5	9.1
	Empl Since Dec	30.9	40.9
	Install Rate Inc	0.0	0.0
	Install Rate Dec	0.0	0.0
	Job Inc	47.9	0.0
	Job Dec	2.0	0.0
MNIST	Blur	15.6	n/a
SpeechCommand	WNoise	14.2	n/a
HotelReview	Pos-Del	861.1	n/a
	Pos-Add	876.1	n/a
	Neg-Del	756.2	n/a
	Neg-Add	794.6	n/a
LunarLander	Relax	124.5	n/a
	Unrelax	3334.5	n/a
BipedalWalker	Relax	290.5	n/a
	Unrelax	153.0	n/a

3.5 Related Work

Sousa and Dillig introduce Cartesian Hoare Logic for verifying k-safety hyperproperties of programs Sousa and Dillig (2016). It was later observed that hyperproperties can also be used to specify ML models (*e.g.*, Seshia *et al.* (2018); Sharma and Wehrheim (2020); Fluri *et al.* (2024)). 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); Zeng *et al.* (2023); Wang *et al.* (2022)), fairness (*e.g.*, Albarghouthi *et al.* (2017); Bastani *et al.* (2019); Urban *et al.* (2020); Khedr and Shoukry (2023)), and others (*e.g.*, Katz *et al.* (2017); Wang *et al.* (2018); Banerjee *et al.* (2024)). 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); Luu *et al.* (2024), object detection Zhou and Sun (2019), and translation He *et al.* (2020).

Beyond robustness and fairness 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.* (2021), Deng *et al.* (2022) and Luu *et al.* (2024) also focus on specifying and testing monotonicity properties in autonomous driving. 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 theses works targets general hyperproperties. Fluri *et al.* (2024) study evaluating ML models' *superhuman* capabilities via consistency checks similar to our hyperproperties. Later, Li *et al.* (2024) proposes a method for improving consistency of large language models.

In this work, 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.

3.6 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. Although users could manually write test cases or a test harness for each desired property, this would be tedious, repetitive, and easy to 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.

Chapter 4

Automatically Testing Functional Properties of Code Translation Models

Large language models are becoming increasingly practical for translating code across programming languages, a process known as *transpiling*. Even though automated transpilation significantly boosts developer productivity, a key concern is whether the generated code is correct. Existing work initially used manually crafted test suites to test the translations of a small corpus of programs; these test suites were later automated. In contrast, we devise the first approach for automated, functional, property-based testing of code translation models. Our general, user-provided specifications about the transpiled code capture a range of properties, from purely syntactic to purely semantic ones. As shown by our experiments, this approach is very effective in detecting property violations in popular code translation models, and therefore, in evaluating model quality with respect to given properties. We also go a step further and explore the usage scenario where a user simply aims to obtain a correct translation of some code with respect to certain properties without necessarily being concerned about the overall quality of the model. To this purpose, we develop the first property-guided search procedure for code translation models, where a model is repeatedly queried with slightly different parameters to produce alternative and potentially more correct translations. Our results show that this search procedure helps to obtain significantly better code translations.

4.1 Introduction

Large language models (LLMs) are becoming highly relevant for translating code across programming languages, a process also known as *transpiling*. Transpilation is typically used to translate an existing software system written in an obsolete programming language into a modern language or to integrate code bases written in different languages into one. On one hand, automated transpilation tremendously increases developer productivity. On the other hand, a key concern is whether the transpiled code is correct.

None of the existing work on code translation Rozière *et al.* (2020); Lachaux *et al.* (2021); Rozière *et al.* (2022); Szafraniec *et al.* (2023); Tang *et al.* (2023); Pan *et al.*

(2024); Ibrahimzada *et al.* (2024); Eniser *et al.* (2024b); Jamsaz *et al.* (2024); Macedo *et al.* (2024); Shetty *et al.* (2024); Saha *et al.* (2024) offer a widely applicable, structured methodology for validating the correctness of translations, instead they rely on ad-hoc evaluation approaches.

Our approach. In this work, we automatically test functional properties of code translation models themselves. To this end, we extend NOMOS Christakis *et al.* (2023), an open-source framework for expressing functional-correctness properties of machine learning models and automatically testing models against these properties. In particular, NOMOS uses a declarative, domain-agnostic specification language for writing *hyperproperties* Clarkson and Schneider (2008) (or *k-safety properties*), which capture functional correctness by reasoning about k model executions. As an example, consider a recidivism-risk model predicting whether a criminal is likely to re-offend. The property that "if a criminal's number of priors increases, then their recidivism risk should not decrease" is a *2-safety property*—we need two model executions to detect a violation of this property, both of which take as input the same criminal but one with an increased number of priors. NOMOS has been used to effectively test models from various application domains, namely action policies as well as models that take as input tabular data, images, speech, and natural language.

Here, we build on NOMOS to enable it to express and validate a wide range of k-safety properties about code translation models, ranging from purely *syntactic* to purely *semantic* properties of the transpiled code. An example of a purely syntactic property is that "the number of loops in the translated code should match the number of loops in the original code"; a purely semantic property is that "the translated code should produce the same return values as the original code for a given set of inputs". We can also express *compilation preservation*, that is, "if the original code compiles, the translated code should also compile". Note that compilers typically perform both syntactic (e.g., parsing) and semantic (e.g., type checking) analyses, and thus, compiler preservation could be viewed as a hybrid property, between purely syntactic and purely semantic ones. These are examples of standard (1-)safety properties—they can be validated by a single model execution that generates the translated code.

A k greater than 1 enables specifying the expected model behavior more comprehensively. For instance, violations of the above compilation property may be unavoidable for some programs—e.g., the source program uses a library function for which there is no comparable version in the target language. Writing k-safety properties allows not labeling such unavoidable model behavior as "buggy" and identifying more severe issues. An example of such a 2-safety property is: "given a program P, if a function parameter is renamed in P' and compilation preservation holds for one of the two programs, then it should also hold for the other". More specifically, P' is an equivalent variant of P that is randomly generated by renaming a function parameter in P. A property violation is de-

tected when P' (conversely, P) and its translated version compile whereas the translated version of P (conversely, P') does not compile. Such a violation indicates more severe buggy behavior of the model than if, say, P compiled but its translated version did not (1-safety). After all, we know that compilation preservation must hold when applying a simple, equivalent transformation.

Properties like these can be succinctly expressed and validated in our NOMOS extension. On a high level, the user provides a code translation model and the (k-safety) properties that it should satisfy. The output is a set of tests that violate the given properties. For example, for the above 2-safety property, program P would be selected by NOMOS from an existing corpus, such as the model test set, and P' would be automatically generated. As output, NOMOS would produce tests, each comprising 2 inputs to the model, namely P and P', for which the property fails. Under the hood, we automatically translate the given properties into a *test harness*, that is, code that uses *metamorphic testing* Chen *et al.* (1998); Segura *et al.* (2016) to generate inputs to the model under test and validate its outputs against an oracle expressing the expected behavior.

As our results show, our approach is very effective in detecting property violations in popular code translation models, such as TRANSCODER Rozière *et al.* (2020), DOBF Lachaux *et al.* (2021), TRANSCODER-IR Szafraniec *et al.* (2023), and STARCODER Li *et al.* (2023a). When testing these models against 38 properties that we specified, we detect thousands of violations. When used in this way, our approach can therefore help evaluate the quality of the models under test with respect to given properties. Any detected violations could even help repair the models although it could be costly Ouyang *et al.* (2022).

In this chapter, we explore another usage scenario in which the user aims to obtain a correct translation of some code with respect to a set of properties without updating the model. As a result, we devise a property-guided search procedure for code translation models, where a model is repeatedly queried with slightly different parameters (e.g., temperature) to produce an alternative translation that potentially satisfies the desired properties. Note that, for this scenario, there are certain guidelines for writing k-safety properties that are compatible with our search (see Section 4.3).

In summary, we make the following contributions:

- We devise the first approach for automated, functional, property-based testing of code translation models, which we implement as an extension of NOMOS. Our implementation is publicly available[1].
- We present the first formalization of k-safety properties for this domain, ranging

[1] https://github.com/Rigorous-Software-Engineering/nomos

from purely syntactic to purely semantic ones.

- We develop the first property-guided search procedure for code translation models to generate alternative translations that potentially satisfy a given set of properties.
- We evaluate the effectiveness of our approach in detecting violations of 38 properties across four state-of-the-art code translation models. We also show that our search procedure can help obtain significantly better translations for a given user-provided program.

4.2 Related Work

Code translation. Earlier works on *code translation* Rozière *et al.* (2020); Lachaux *et al.* (2021); Rozière *et al.* (2022); Szafraniec *et al.* (2023) use task-specific language models and rely on manually written tests to evaluate the semantic correctness of translated code (similar to our semantic 1-safety property) on relatively small, curated program corpora. Among these, Rozière *et al.* (2022) stands out by employing automatically generated tests. With the rise of general-purpose LLMs, recent works on code translation Tang *et al.* (2023); Pan *et al.* (2024); Ibrahimzada *et al.* (2024); Eniser *et al.* (2024b); Jamsaz *et al.* (2024); Macedo *et al.* (2024); Shetty *et al.* (2024); Saha *et al.* (2024) mostly leverage off-the-shelf LLMs. However, none of these studies offer a structured methodology for validating the correctness of translations, instead relying on ad-hoc evaluation approaches. In contrast, we provide a comprehensive framework for testing the correctness of models and translations. Our approach enables NOMOS to express a broader and more extensive set of correctness properties and automatically generate new programs, rather than depending solely on curated corpora.

Code generation. There is also work based on LLMs for *code generation* that uses prompts in (primarily) natural language and evaluates the correctness of the generated code. HUMANEVAL Chen *et al.* (2021) is a popular benchmark in this context, but it uses a small number of tests to evaluate the semantic correctness of the generated code (similar to our semantic 1-safety property). EVALPLUS Liu *et al.* (2023) extends HUMANEVAL to obtain more comprehensive benchmarks—it uses fuzzing to automatically generate many more tests. Other work Cassano *et al.* (2023); Athiwaratkun *et al.* (2023) has proposed methods for extending benchmarks, such as HUMANEVAL and MBPP Austin *et al.* (2021), to more programming languages. ReCode Wang *et al.* (2023) checks robustness properties (somewhat similar to our semantic 2-safety properties, but for code generation instead of code translation) by slightly perturbing the prompts through over 30 transformations. Gu *et al.* (2024) presents a new semi-automatically created benchmark consisting of 800 Python functions for evaluating code understanding and reasoning of LLMs.

Constraining LLM outputs. Constraining LLM outputs to enforce certain validity criteria has also been explored. For instance, in the context of programming languages, such criteria may enforce syntactic or semantic constraints on the output programs or completions Scholak *et al.* (2021); Poesia *et al.* (2022). On a high level, our search procedure pursues a similar goal in the context of code translation, but phrases it as an optimization problem that aims to minimize the number of violated properties. More generally, such validity criteria can also consist of a grammar Shin *et al.* (2021) or a domain-specific query language Beurer-Kellner *et al.* (2023). Saha *et al.* (2024) augments translation prompts with LLM-generated natural language specifications in an attempt to constrain outputs.

4.3 Approach

In this section, we first give an overview of our specifications for code translation models through examples (Section 4.3.1). We then describe our testing (Section 4.3.2) and search (Section 4.3.3) procedures in detail.

4.3.1 Specifications

We introduce our extended NOMOS specification language through four example properties, namely, two 1-safety properties (a syntactic and a semantic one) and two 2-safety properties (again, a syntactic and a semantic one). On a high level, each specification typically consists of:

- A *precondition*, which expresses the conditions under which the model under test should be called;
- A block of arbitrary source code (written in Python), which invokes the model under test;
- A *postcondition*, which expresses the safety property that the model should satisfy.

We presented a formal description of the language in Chapter 3. We describe the extensions in the next section.

First, consider the syntactic 1-safety property shown in Figure 4.1a expressing that, when transpiling Java code into C++, the number of conditionals in the input (Java) program should match the number of conditionals in the output (C++) program. Section 4.3 declares the input program `pj` and section 4.3 the corresponding output program `pc`. These declarations are followed by the block of Python code (within curly braces), which invokes the model under test to transpile `pj` and assigns the resulting code to `pc`—see lines 3–5. On line 6, the `ensures` clause expresses the postcondition that the number of

```
input pj;
output pc;
{
    pc = transpile(pj, "java", "cpp")
}
ensures numConditionals(pj, "java") == numConditionals(pc, "cpp");
```

(a) Syntactic 1-safety property.

```
input pj;
requires compiles(pj, "java");
output pc;
{
    pc = transpile(pj, "java", "cpp")
}
ensures compiles(pc, "cpp")
    ==> retValues(pj, "java") == retValues(pc, "cpp");
```

(b) Semantic 1-safety property.

```
input pj1;
var pj2 := addConditional(pj1, "java");
output pp1;
output pp2;
{
    pp1 = transpile(pj1, "java", "py")
    pp2 = transpile(pj2, "java", "py")
}
ensures numLoops(pp1, "py") == numLoops(pp2, "py");
```

(c) Syntactic 2-safety property.

```
input pj1;
var pj2 := renameParam(pj1, "java");
requires pj2 != null;
output pp1;
output pp2;
{
    pp1 = transpile(pj1, "java", "py")
    pp2 = transpile(pj2, "java", "py")
}
ensures compiles(pp1, "py") && compiles(pp2, "py")
    ==> retValues(pp1, "py") == retValues(pp2, "py");
```

(d) Semantic 2-safety property.

Figure 4.1: Example k-safety specifications for code translation models.

conditionals in `pj` should be equal to the number of conditionals in `pc`. Note that there is no precondition in this property, i.e., the precondition is true.

Figure 4.1b shows a semantic 1-safety property expressing, with a precondition on section 4.3, that the model should be called with a compiling Java program. Note that preconditions are specified with `requires` clauses. The postcondition says that, if the resulting C++ program is also compiling, then the return values of the two programs should match. In other words, given the same input values, the two programs should return the same output values.

A syntactic 2-safety property about a Java-to-Python translation is shown in Figure 4.1c. On section 4.3, the property declares an input program `pj1`. Unlike the previous properties, it also declares an additional program `pj2` on section 4.3—`pj2` is generated by adding a random conditional to `pj1` such that the input/output behavior of the code remains unaffected. For instance, the body of the conditional may just consist of a print statement. On lines 3–4, the property declares *two* output programs `pp1` and `pp2`, which are assigned by the *two* model invocations (lines 6–7). The postcondition checks that the number of loops in `pp1` and `pp2` matches.

Finally, consider the semantic 2-safety property in Figure 4.1d. Here, `pj2` is generated from `pj1` by renaming a random function parameter (section 4.3). The precondition expresses that the model should be invoked if the renaming succeeds. The postcondition checks that the return values of the two output programs match provided that both compile—in Python, this means that both programs parse.

In this work, we specified a total of 62 properties—see Appendices B.2 and B.3.

4.3.2 Testing Procedure

Our testing procedure for these properties is based on the existing NOMOS framework Christakis *et al.* (2023). Internally, the NOMOS framework generates a Python test harness for the given model and its specification. The harness tests the model until a user-specified budget is depleted. Specifically, for each budget unit, the harness generates inputs for the model such that any precondition is satisfied, executes the block of Python code in the specification, and checks the postcondition. Finally, the NOMOS framework records, processes, and de-duplicates all detected property violations.

The harness essentially implements *metamorphic testing* Chen *et al.* (1998); Segura *et al.* (2016), which constitutes a natural choice for checking k-safety properties. Given an input to a system under test (in our case, a model under test), metamorphic testing transforms the input such that the relation of the corresponding outputs is known. For instance, given a criminal as input to a model that predicts recidivism risk, a metamorphic

transformation could increase the criminal's number of priors (keeping all other attributes the same). Then, we know that the recidivism risk of the new criminal should be at least as high as that of the original one. Similarly, a NOMOS property for $k > 1$ also describes input transformations, and the expected relation among outputs is the postcondition—see Figure 4.1.

To support code translation models, we extended NOMOS to enable expressing and testing their properties. First, we incorporated two new classes of domain-specific functions, namely, *program-transformation* and *program-inspection* functions. In Figure 4.1, we use transformation functions `addConditional` and `renameParam`, and inspection functions `numConditionals`, `numLoops`, `compiles`, and `retValues`. In total, we added 7 transformation and 5 inspection functions to express our properties—see Appendix B.1.

LLMs generate their outputs token by token, and by default, the next token is selected greedily by returning the most probable token. The main drawback of this greedy search is that there is a chance of missing high-probability tokens that are hidden behind lower-probability ones, thus generating sub-optimal predictions. A beam size of N tokens alleviates this issue by selecting the most probable N tokens at every step, and in the end, generating N predictions. We, thus, extended NOMOS to allow enabling a beam size of N, which for a k-safety property means that we have k model queries each producing N predictions. When generating the failing tests, NOMOS only reports inputs for which all N^k prediction combinations violate the property.

Given that LLMs are expensive to query, we also added a caching mechanism to avoid the cost of asking the same queries repeatedly and thus slowing down our testing procedure. More specifically, due to randomness in the program-transformation functions, we might generate the same inputs for a model under test when checking different (or even the same) properties. We, therefore, cache model queries and the corresponding outputs. Note that this also helps to avoid inconsistent outputs in the case of stochastic models.

Finally, we extended the harness generator to allow users to control different model parameters, like the temperature.

4.3.3 Search Procedure

Being able to check functional properties of code translation models opens up a new use case, namely one where the user only aims to generate a correct translation of a piece of code without necessarily testing the overall model quality. To address this use case, we developed a property-guided search procedure that repeatedly queries a model with slightly different parameters (such as the temperature) to produce alternative and potentially more correct translations with respect to the given properties. In other words, our

search procedure takes as input an initial model instance (with user-provided parameters) and searches for model instances (same model but with different parameters) that can satisfy more properties for the particular piece of code.

On a high level, the search procedure repeatedly invokes the testing procedure with mutated model parameters to optimize the number of violated properties. It returns the model output as soon as all properties are satisfied by the current model instance. If all properties cannot be satisfied within a given search budget, it returns the best model output, which results in the fewest violated properties.

Algorithm 6 describes the search procedure more precisely. It takes a set of properties, a program to be translated, a search budget, a test budget, and the initial model parameters; it returns the best program translation found. The while-loop on line 6 iterates until the search budget is depleted. Each iteration runs the testing procedure for all properties with the current model parameters and calculates the number of violated properties and the total number of property violations (line 9). If no properties are violated, the current translation is returned (lines 14–15). Otherwise, on line 16, we use a lexicographic fitness function to minimize the number of violated properties before minimizing the total number of violations. For the next iteration, line 20 mutates the current model parameters. This essentially performs stochastic hill climbing, but more sophisticated optimization techniques could easily be used instead.

For this use case, each k-safety property should require a single input (P_1) from the user and generate the remaining inputs ($P_2, \ldots, P_k$) automatically. For example, in Figure 4.1c, the user-provided input is `pj1`, whereas `pj2` is generated from `pj1`. Then, our search procedure will optimize the translation of P_1.

In addition, note that a violation of a k-safety property could be caused by sub-optimal model performance for any of the k model invocations. However, for this use case, the user is only interested in the model behaving as expected for P_1, ignoring any violations caused only by its variants $P_2, \ldots, P_k$. This preference can be encoded directly in the property by ensuring that a violation occurs only if the user-provided input is to blame. In particular, each property should only generate equivalent or "harder" variants of P_1, where "harder" means containing additional code for translation. The postcondition should then express that, if the model succeeds for (potentially harder) $P_2, \ldots, P_k$, then it should also succeed for (potentially easier) P_1. When this postcondition is violated, we know that the model output is sub-optimal for the user-provided input. For instance, to make the property of Figure 4.1c compatible with our search procedure, we could change the postcondition to:

```
ensures numLoops(pp2, "py") == numLoops(pj2, "java")
    ==> numLoops(pp1, "py") == numLoops(pj1, "java");
```

Algorithm 6: Search procedure

```
1  Function SEARCH(properties, program P, searchBudget, testBudget,
     initModelParams):
2      params ← initModelParams;
       /* Minimum number of violated properties                    */
3      minVP ← +∞;
       /* Minimum number of total violations                       */
4      minTV ← +∞;
5      bestTranslation ← null;
6      while searchBudget > 0 do
           /* Current number of violated properties               */
7          VP ← 0;
           /* Current number of total violations                  */
8          TV ← 0;
           /* Run the testing procedure for each property with current model
              parameters                                           */
9          foreach prop in properties do
               /* Test returns violations and program translation */
10             (v, tr) ← Test(prop, P, testBudget, params);
11             TV ← TV + v;
12             if v > 0 then
13                 VP++;
14         if VP == 0 then
15             return tr;
16         if (VP < minVP) or (VP == minVP and TV < minTV) then
17             minVP ← VP;
18             minTV ← TV;
19             bestTranslation ← tr;
20         params ← Mutate(params);
21         searchBudget ← searchBudget - 1;
22     return bestTranslation;
```

Table 4.1: The percentage of property violations (i.e., unique failing tests / total number of tests x 100%) detected when running the testing procedure on TRANSCODER and TRANSCODER-IR for translating from Java to C++. The first column (BS) shows the beam-size parameter, and the second column the value of k of the corresponding k-safety property. The third column shows the (abbreviated) program-transformation (PT) functions, which are combined with the (abbreviated) program-inspection (PI) functions of the first row to form the properties in Appendix B.2. We report the percentage of violations under each PI function—the left sub-column shows the percentage of violations for TRANSCODER and the right for TRANSCODER-IR.

BS	k	PT \ PI	arity		numC/s		numL/s		compiles		retV/s	
1	1	–	0	0	<1	<1	0	<1	19	22	51	47
	2	rnmP	0	<1	<1	<1	0	<1	14	9	8	19
		addP	0	0	<1	<1	<1	<1	34	36	20	24
		addC	0	0	<1	<1	0	<1	40	17	45	26
		chBC	0	0	<1	<1	<1	<1	52	54	–	–
		addL	0	0	<1	<1	<1	<1	37	28	15	39
		rmL	0	0	0	0	<1	0	2	10	–	–
	3	mrg	<1	<1	<1	<1	<1	<1	61	61	2	2
3	1	–	0	0	0	0	0	<1	1	1	5	5
	2	rnmP	0	0	<1	<1	0	0	< 1	4	0	< 1
		addP	0	0	0	<1	0	<1	< 1	2	9	7
		addC	0	0	0	<1	0	0	< 1	2	8	8
		chBC	0	0	<1	<1	<1	<1	4	17	–	–
		addL	0	0	<1	<1	<1	0	< 1	1	< 1	18
		rmL	0	0	0	0	0	0	< 1	< 1	–	–
	3	mrg	<1	<1	<1	<1	<1	<1	12	24	0	< 1

4.4 Evaluation

4.4.1 Experimental Setup

Models. For our experiments, we use the pre-trained models TRANSCODER Rozière *et al.* (2020), DOBF Lachaux *et al.* (2021), TRANSCODER-IR Szafraniec *et al.* (2023), and STARCODER Li *et al.* (2023a). The first three expect a function as input and predict the corresponding function in the output language. For STARCODER, we use completion mode and send queries that provide an input function and request the output function to be completed. We evaluate TRANSCODER for both Java-to-C++ and Java-to-Python translations. TRANSCODER-IR is evaluated for Java to C++ (it was not trained

Table 4.2: The percentage of property violations detected when running the testing procedure on TRANSCODER, DOBF, and STARCODER for translating from Java to Python. Under each PI function, the left sub-column shows the percentage of violations for TRANSCODER, the middle for DOBF, and the right for STARCODER.

BS	k	PI / PT	arity			numC/s			numL/s			compiles			retV/s		
1	1	–	0	0	<1	2	<1	3	2	3	3	21	17	6	42	47	59
	2	rnmP	0	0	0	1	1	3	2	<1	38	8	7	4	5	6	17
		addP	0	0	<1	2	1	4	3	1	4	7	8	47	5	5	14
		addC	0	<1	<1	1	<1	86	2	1	3	5	9	12	25	17	8
		chBC	0	0	<1	2	<1	19	3	1	4	43	47	19	–	–	–
		addL	0	0	<1	<1	1	5	<1	2	42	8	8	11	25	26	25
		rmL	0	0	<1	<1	0	3	1	<1	4	4	5	3	–	–	–
	3	mrg	0	<1	<1	7	4	10	9	4	7	40	45	19	5	6	21
3	1	–	0	0	0	<1	0	< 1	<1	<1	< 1	3	2	1	5	7	12
	2	rnmP	0	0	0	<1	<1	< 1	<1	<1	< 1	< 1	< 1	< 1	< 1	< 1	1
		addP	0	0	<1	0	<1	< 1	<1	<1	< 1	< 1	< 1	3	< 1	< 1	2
		addC	0	0	0	0	0	82	<1	<1	1	< 1	< 1	8	15	8	3
		chBC	0	0	0	<1	<1	11	<1	<1	1	28	35	15	–	–	–
		addL	0	0	0	<1	<1	< 1	<1	<1	23	< 1	< 1	4	14	17	19
		rmL	0	0	0	0	0	< 1	<1	<1	3	2	2	1	–	–	–
	3	mrg	0	0	0	<1	<1	3	1	<1	2	15	22	56	< 1	2	9

on Python) and DOBF for Java to Python (it was not trained on C++). We evaluate STARCODER for Java to Python since it was fine-tuned for Python.

Program benchmarks. We use the benchmark set introduced in TRANSCODER Rozière *et al.* (2020), including 545 solutions to LeetCode problems implemented in Java, C++, and Python. On average, each program comes with ca. 10 tests that specify the expected output values for given input values. (These are the input values used by `retValues`.)

Parameters. We run the testing procedure with beam size 1 and 3 and set the temperature to 0.1. For the search, the beam size is 1, and we mutate the model temperature, which is initially 0.1.

4.4.2 Results for Testing Procedure

We evaluate our testing procedure by checking a total of 38 properties found in Appendix B.2.

We introduced many inspection and transformation functions; we now provide a sum-

Table 4.3: The total number of violations (TV), the number of violated properties (VP), and the number of violated syntactic properties (VSP) detected when running the testing procedure on all models with beam size (BS) 1 and 3.

BS		TRANSCODER Java-C++	TRANSCODER-IR Java-C++	TRANSCODER Java-Python	DOBF Java-Python	STARCODER Java-Python
1	TV	8663	8603	5777	5564	11213
	VP	27	30	30	31	37
	VSP	13	16	16	17	23
3	TV	1035	2326	2240	2535	6611
	VP	20	25	27	26	31
	VSP	8	11	13	13	17

mary of the remaining ones (see Appendix B.1 for details). The `arity` inspection function returns the number of parameters of a given function. The `addParam` transformation function adds a random parameter to the function without affecting its input/output behavior, and `addLoop` adds a random for-loop. The `rmLoop` function removes a random for-loop from the function, while `chBranchCond` randomly changes the branch condition of an if- or switch-statement. Both `rmLoop` and `chBranchCond` may change the input/output behavior of the original code and are thus not used in semantic properties. Finally, `merge` merges two functions by executing one of them depending on an additional Boolean argument.

In our experiments, we set the testing budget to 500 for all 1-safety properties and to 2500 for all other properties. All 545 programs from our benchmark set may be used as inputs when testing the models. Note that the testing budget for 1-safety properties is lower since these properties only select a single program from the corpus and invoke the model under test on this program; thus, a higher budget would unlikely result in a significantly higher number of unique violations.

Next, we present our results for the testing procedure organized in the following five research questions (RQs).

RQ1. Is the testing procedure effective in detecting property violations? Tables 4.1 and 4.2 show the percentages of (unique) property violations for all models. The absolute numbers are included in Appendix B.4 but are summarized in Table 4.3.

When using a beam size of 1, our testing procedure finds between 27 (TRANSCODER for Java to C++) and 37 (STARCODER for Java to Python) violated properties (out of 38). The number of total property violations ranges from 5564 (DOBF for Java to Python) to 11213 (STARCODER for Java to Python). Our testing procedure is, therefore, highly effective in detecting violations.

When increasing the beam size to 3, we observe a reduced number of violations and violated properties. In particular, the number of violated properties decreases by between 3 (TRANSCODER for Java to Python) and 7 (TRANSCODER for Java to C++). This confirms that increasing the beam size can improve the quality of the translations.

RQ2. How do models perform with respect to different properties? As shown in the tables, the models generally perform better for purely syntactic properties, i.e., using the `arity`, `numConditionals`, and `numLoops` inspection functions. In particular, for a beam size of 1, the number of violated syntactic properties ranges from 13 (TRANSCODER for Java to C++) to 23 (STARCODER for Java to C++) (out of 24), whereas the number of other violated properties (i.e., using `compiles` and `retValues`) is 14 (out of 14) for all models. For a beam size of 3, the number of violated syntactic properties decreases for all models, and more specifically, by between 3 (TRANSCODER for Java to Python) and 6 (STARCODER for Java to Python). In contrast, the number of other violated properties only decreases by 2 for TRANSCODER for Java to C++ and by 1 for DOBF for Java to Python.

RQ3. How does TRANSCODER perform with respect to the target language? We only evaluate TRANSCODER for two target languages, namely C++ and Python. It generally performs better for C++, where it violates 27 (out of 38) properties for a beam size of 1 and 20 for a beam size of 3. For Python, it violates 30 and 27 properties, respectively. This not unexpected since C++ is more similar to Java than Python. Interestingly however, for TRANSCODER for C++, we detect a total of 6084 violations of `compiles` properties in contrast to 2996 violations for Python (for a beam size of 1). Again, this is not unexpected since these properties only check program parsing for Python.

RQ4. How do translation models perform in comparison to the general-purpose model STARCODER? We found significantly more property violations for STARCODER than for the specialized translation models. In particular, for a beam size of 1, we detected 37 (out of 38) violated properties and 11213 total violations for STARCODER, 30 violated properties and 5777 total violations for TRANSCODER, and 31 violated properties and 5564 total violations for DOBF. This is not surprising given that the specialized models are specifically trained for translation.

RQ5. What is the average running time for checking a property on a given model? The average running time (including model inference and test harness execution) for checking a property ranges between 1.3s (DOBF for Java to Python) and 42.9s (STARCODER for Java to Python) for beam size 1. When increasing the beam size to 3, the running time increases slightly for all models (for instance, to 3.3s for DOBF for Java to Python and to 51.3s for STARCODER for Java to Python). We include the average running time for all models in Appendix B.4.

These experiments were run on cluster machines with A100 Nvidia Tesla GPUs and Intel Xeon Gold 5317 CPUs, running Debian GNU/Linux 11. Each GPU has 80GB memory allowing to host the larger StarCoder model.

4.4.3 Results for Search Procedure

We evaluate the effectiveness of our search procedure for TransCoder and DOBF by generating Java-to-Python translations for 100 randomly selected programs from our benchmark set. We check 24 search properties (see Appendix B.3) for each translation.

We set the search budget to 20 and the testing budget to 50 (see Algorithm 6). We use relatively small budgets to keep the running time small. We start the search with an initial temperature of 0.1 and mutate it at every iteration by adding Gaussian noise with $\mu = 0$ and $\sigma^2 = 0.01$.

RQ1. Is the search procedure effective in finding better translations? Figure 4.2 (top) shows the number of "passing programs" (i.e., programs for which the testing procedure reports no violations) out of 100 along the y-axis as we increase the number of search iterations from 1 to 20 along the x-axis. As shown in the figure, the search significantly increases the number of passing programs (from 66 to 84 for TransCoder and from 71 to 83 for DOBF). It also reduces the number of violated properties for the final translation by almost half (from 78 to 35 for TransCoder and from 100 to 63 for DOBF). Our search procedure is, therefore, effective in improving the quality of the translation with respect to the number of violated properties.

RQ2. How long does the search take? As shown in Figure 4.2 (top), exploring only a few (e.g., 6) model instances leads to significant improvements. The mean number of search iterations is 4.5 for TransCoder and 4.7 for DOBF.

RQ3. How does the search affect the number of violations of syntactic versus other properties? Figure 4.2 (bottom) shows how the search affects the number of violations of syntactic and other properties (i.e., using `compiles` and `retValues`). Interestingly, the search helps the most in decreasing the number of violations of other properties, which are the hardest to satisfy.

4.5 Conclusion

In this work, we introduced the first approach for automatically testing user-provided, functional properties of code translation models. We extended the Nomos framework

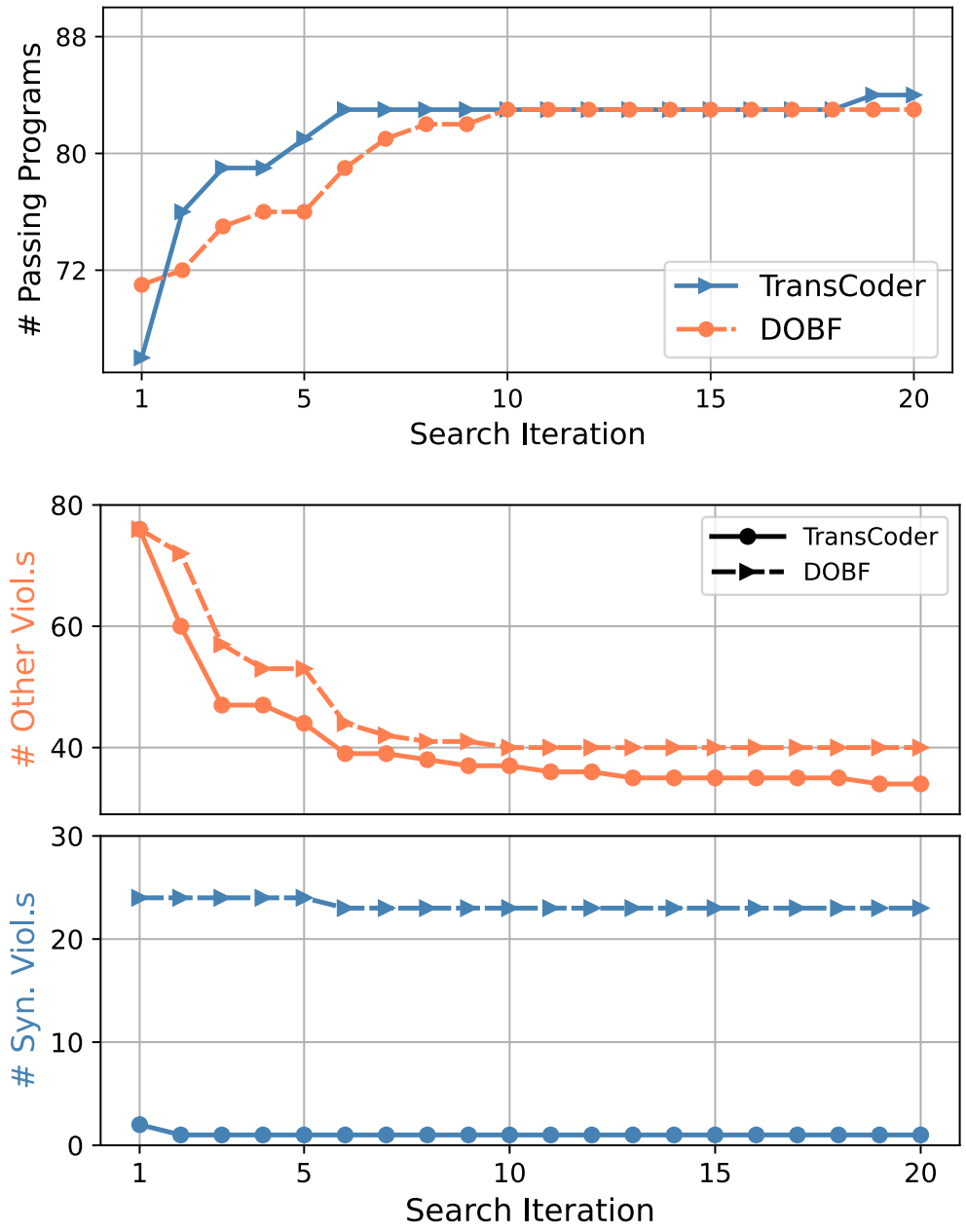


Figure 4.2: The number of passing programs (top) and the number of syntactic and other property violations (bottom) as the number of search iterations increases from 1 to 20.

with domain-specific functions to formalize a broad set of 38 properties, which we evaluated by testing four popular models. We also introduced a property-guided search procedure that aims to optimize the model output based on the number of violated properties.

In future work, we plan to transfer this idea to other settings to effectively enforce quality criteria on model outputs.

Chapter 5

Conclusion and Future Work

This thesis makes a step towards addressing the critical need for reliable and safe machine learning (ML) models, particularly as they become integral to safety-critical domains like autonomous systems and software engineering. Starting with our foundational work π-fuzz Eniser *et al.* (2022) we made a significant step towards testing sequential decision making policies. Our approach involved crafting novel metamorphic test oracles and adapting a widely used software testing technique, fuzzing, for effective evaluation of such policies.

Building on this foundation, we later formalized the concept of k-safety properties to specify and test functional correctness in ML models Christakis *et al.* (2023). Through the introduction of NOMOS, a declarative and domain-agnostic specification language, we provided a widely applicable tool to validate complex hyperproperties across a range of domains, including image classification, speech recognition and action policies. This work highlighted the potential of structured specifications and automated testing frameworks to generalize functional correctness testing across diverse ML artifacts. We have identified thousands of property violations across various domains with NOMOS.

Finally, we extended these methodologies to the domain of code translation models, addressing the unique challenges of testing ML-driven software engineering tools Eniser *et al.* (2024a). We developed two classes of functions which enable forming varying NOMOS properties for testing code translation. Furthermore, we present a procedure for obtaining better translation in terms of the set of specified properties. Our extension allowed us to identify thousands of translation errors in various code translation models.

These contributions reflect the overarching goals of this thesis: to introduce practical tools and techniques inspired by software testing principles, enabling the systematic assessment of ML model reliability and safety. These efforts are stepping stones to achieve the broader objective of addressing the challenges posed by the deployment of ML systems in real-world applications where failure comes with a great cost. We make these tools and techniques publicly available to enable their broader usage.

The techniques presented in this dissertation stand out for their practicality. Looking ahead, several directions could further enhance their utility. One promising direction is the automatic derivation of NOMOS specifications from positive and negative examples of model predictions. Additionally, integrating NOMOS with formal verification techniques, where feasible, could offer stronger correctness guarantees, especially in high-stakes domains like autonomous systems and critical infrastructure. Furthermore, property violations provide valuable insights into the weaknesses of the model under test. Going one step further to address these weaknesses, the NOMOS testing procedure could generate a dataset for further model training or fine-tuning based on passed and violated properties. While a feasibility study has been conducted on Chapter 2, this approach requires further exploration in other domains.

Appendix A

Appendix for Chapter 3

A.1 Complete List of Specifications

COMPAS—*Felony Inc.* If the number of committed felonies for a criminal increases, then their recidivism risk should not decrease.

COMPAS—*Felony Dec.* If the number of committed felonies for a criminal decreases, then their recidivism risk should not increase.

COMPAS—*Misdmnr Inc.* If the number of committed misdemeanors for a criminal increases, then their recidivism risk should not decrease.

COMPAS—*Misdmnr Dec.* If the number of committed misdemeanors for a criminal decreases, then their recidivism risk should not increase.

COMPAS—*Priors Inc.* If the number of priors for a criminal increases, then their recidivism risk should not decrease.

COMPAS—*Priors Dec.* If the number of priors for a criminal decreases, then their recidivism risk should not increase.

COMPAS—*Others Inc.* If the number of other crimes committed by a criminal increases, then their recidivism risk should not decrease.

COMPAS—*Others Dec.* If the number of other crimes committed by a criminal decreases, then their recidivism risk should not increase.

COMPAS—*IsRecid Set.* If a criminal becomes a recidivist, then their recidivism risk should not decrease.

COMPAS—*IsRecid Unset.* If a criminal ceases to be a recidivist, then their recidivism risk should not increase.

COMPAS—*IsVRecid Set.* If a criminal becomes a violent recidivist, then their recidivism risk should not decrease.

COMPAS—*IsVRecid Unset.* If a criminal ceases to be a violent recidivist, then their recidivism risk should not increase.

GermanCredit—*Crdt Amount Inc.* If the credit amount requested by a person increases, then they should not be more likely to receive it.

GermanCredit—*Crdt Amount Dec.* If the credit amount requested by a person decreases, then they should not be less likely to receive it.

GermanCredit—*Crdt Hist Inc.* If a person's credit history worsens, then they should not be more likely to receive credit.

GermanCredit—*Crdt Hist Dec.* If a person's credit history improves, then they should not be less likely to receive credit.

GermanCredit—*Empl Since Inc.* If a person's employment years increase, they should not be less likely to receive credit.

GermanCredit—*Empl Since Dec.* If a person's employment years decrease, they should not be more likely to receive credit.

GermanCredit—*Install Rate Inc.* If a person's installment rate (as a percentage of their disposable income) increases, then they should not be more likely to receive credit.

GermanCredit—*Install Rate Dec.* If a person's installment rate (as a percentage of their disposable income) decreases, then they should not be less likely to receive credit.

GermanCredit—*Job Inc.* If a person is promoted, then they should not be less likely to receive credit.

GermanCredit—*Job Dec.* If a person is demoted, then they should not be more likely to receive credit.

MNIST—*Blur.* If a blurred image is correctly classified, then its unblurred version should also be correctly classified.

SpeechCommand—*WNoise.* If a speech command with white noise is correctly classified, then its non-noisy version should also be correctly classified.

HotelReview—*Pos-Del.* Deleting the positive comments of a hotel review should not make it more positive.

HotelReview—*Pos-Add.* If more positive comments are added to a hotel review, it should not become more negative.

HotelReview—*Neg-Del.* Deleting the negative comments of a hotel review should not make it more negative.

HotelReview—*Neg-Add.* If more negative comments are added to a hotel review, it should not become more positive.

LunarLander—*Relax.* If the lander lands successfully, then decreasing the surface height (thus giving the lander more time to land) should also result in landing successfully.

LunarLander—*Unrelax.* If the lander fails to land, then increasing the surface height (thus giving the lander less time to land) should also result in failing to land.

BipedalWalker—*Relax.* If the walker successfully reaches the end of the terrain, then making the terrain smoother should also result in successfully reaching the end.

BipedalWalker—*Unrelax.* 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.

Appendix B

Appendix for Chapter 4

B.1 New NOMOS Domain-Specific Functions Introduced

B.1.1 Program-Transformation Functions

`addConditional`: Adds a random conditional in a given function without affecting its input/output behavior.

`addLoop`: Adds a random for-loop in a given function without affecting its input/output behavior.

`addParam`: Adds a random parameter in a given function without affecting its input/output behavior.

`chBranchCond`: Replaces the condition of a random if- or switch-statement in a given function. The new condition is generated randomly, and therefore, the input/output behavior of the transformed function may not match the input/output behavior of the original function.

`merge`: Merges the bodies of two given functions into a single function. The new function takes as input a Boolean parameter in addition to the parameters of the two given functions. The body of the new function contains a conditional that branches on the Boolean parameter, and each branch executes one of the given functions.

`renameParam`: Renames a random parameter in a given function without affecting its input/output behavior.

`rmLoop`: Removes a random for-loop from a given function. As a result, the input/output behavior of the transformed function may not match the input/output behavior of the original function.

B.1.2 Program-Inspection Functions

`arity`: Returns the number of parameters of a given function.

`compiles`: Returns true if the given function compiles, otherwise it returns false.

`numConditionals`: Returns the number of conditionals in a given function.

`numLoops`: Returns the number of loops in a given function.

`retValues`: Returns the output values of a given function. The inputs could be randomly generated or imported from an existing test suite of the model under test.

B.2 Specifications for Testing

We name each of our properties based on the program-transformation and program-inspection functions it uses as follows: <program-transformation function> | <program-inspection function>. Note that a program-transformation function is optional. We describe how each function is implemented in Appendix B.1.

B.2.1 1-Safety Properties

arity. The arity of the output function should be equal to the arity of the input function.

numConditionals. The number of conditionals in the output function should be equal to the number of conditionals in the input function.

numLoops. The number of loops in the output function should be equal to the number of loops in the input function.

compiles. If the input function compiles (resp. does not compile), then the output function should also (resp. not) compile. In other words, compilation should be preserved among the input and output functions.

retValues. Given an input function that compiles, if the output function also compiles, then the return values of the output function should be equal to the return values of the input function.

B.2.2 2-Safety Properties

renameParam|arity. Given input function i_1, input function i_2 is created by renaming a random parameter in i_1. The arity of their corresponding output functions should be equal.

renameParam|numConditionals. Given input function i_1, input function i_2 is created by renaming a random parameter in i_1. The number of conditionals in their corresponding output functions should be equal.

renameParam|numLoops. Given input function i_1, input function i_2 is created by renaming a random parameter in i_1. The number of loops in their corresponding output functions should be equal.

renameParam|compiles. Given input function i_1, input function i_2 is created by renaming a random parameter in i_1. Compilation should be preserved among their corresponding output functions.

renameParam|retValues. Given input function i_1, input function i_2 is created by renaming a random parameter in i_1. If their corresponding output functions compile, then their return values should be equal.

addParam|arity. Given input function i_1, input function i_2 is created by adding a random parameter in i_1. Assuming that o_1 and o_2 are their corresponding output functions, the arity of o_2 should be greater than the arity of o_1.

addParam|numConditionals. Given input function i_1, input function i_2 is created by adding a random parameter in i_1. The number of conditionals in their corresponding output functions should be equal.

addParam|numLoops. Given input function i_1, input function i_2 is created by adding a random parameter in i_1. The number of loops in their corresponding output functions should be equal.

addParam|compiles. Given input function i_1, input function i_2 is created by adding a random parameter in i_1. Compilation should be preserved among their corresponding output functions.

addParam|retValues. Given input function i_1, input function i_2 is created by adding a random parameter in i_1. If their corresponding output functions compile, then their return values should be equal.

addConditional|arity. Given input function i_1, input function i_2 is created by adding a random conditional in i_1. The arity of their corresponding output functions should be equal.

addConditional|numConditionals. Given input function i_1, input function i_2 is created by adding a random conditional in i_1. Assuming that o_1 and o_2 are their corresponding output functions, the number of conditionals in o_2 should be greater than the number of conditionals in o_1.

addConditional|numLoops. Given input function i_1, input function i_2 is created by adding a random conditional in i_1. The number of loops in their corresponding output functions should be equal.

addConditional|compiles. Given input function i_1, input function i_2 is created by adding a random conditional in i_1. Compilation should be preserved among their corresponding output functions.

addConditional|retValues. Given input function i_1, input function i_2 is created by adding a random conditional in i_1. If their corresponding output functions compile, then their return values should be equal.

chBranchCond|arity. Given input function i_1, input function i_2 is created by replacing the condition of a random if- or switch-statement in i_1 with a random condition. The arity of their corresponding output functions should be equal.

chBranchCond|numConditionals. Given input function i_1, input function i_2 is created by replacing the condition of a random if- or switch-statement in i_1 with a random condition. The number of conditionals in their corresponding output functions should be equal.

chBranchCond|numLoops. Given input function i_1, input function i_2 is created by replacing the condition of a random if- or switch-statement in i_1 with a random condition. The number of loops in their corresponding output functions should be equal.

chBranchCond|compiles. Given input function i_1, input function i_2 is created by replacing the condition of a random if- or switch-statement in i_1 with a random condition. Compilation should be preserved among their corresponding output functions.

addLoop|arity. Given input function i_1, input function i_2 is created by adding a random for-loop in i_1. The arity of their corresponding output functions should be equal.

addLoop|numConditionals. Given input function i_1, input function i_2 is created by adding a random for-loop in i_1. The number of conditionals in their corresponding out-

put functions should be equal.

addLoop|numLoops. Given input function i_1, input function i_2 is created by adding a random for-loop in i_1. Assuming that o_1 and o_2 are their corresponding output functions, the number of loops in o_2 should be greater than the number of loops in o_1.

addLoop|compiles. Given input function i_1, input function i_2 is created by adding a random for-loop in i_1. Compilation should be preserved among their corresponding output functions.

addLoop|retValues. Given input function i_1, input function i_2 is created by adding a random for-loop in i_1. If their corresponding output functions compile, then their return values should be equal.

rmLoop|arity. Given input function i_1, input function i_2 is created by removing a random for-loop in i_1. The arity of their corresponding output functions should be equal.

rmLoop|numConditionals. Given input function i_1, input function i_2 is created by removing a random for-loop in i_1. The number of conditionals in their corresponding output functions should be equal.

rmLoop|numLoops. Given input function i_1, input function i_2 is created by removing a random for-loop in i_1. Assuming that o_1 and o_2 are their corresponding output functions, the number of loops in o_2 should be less than the number of loops in o_1.

rmLoop|compiles. Given input function i_1, input function i_2 is created by removing a random for-loop in i_1. Compilation should be preserved among their corresponding output functions.

B.2.3 3-Safety Properties

merge|arity. Given input functions i_1 and i_2, input function i_3 is created by merging i_1 and i_2. Assuming that o_1, o_2, and o_3 are their corresponding output functions, the arity of o_3 should be equal to the arity of o_1 plus the arity of o_2 plus 1.

merge|numConditionals. Given input functions i_1 and i_2, input function i_3 is created by merging i_1 and i_2. Assuming that o_1, o_2, and o_3 are their corresponding output functions, the number of conditionals in o_3 should be equal to the number of conditionals in o_1 plus the number of conditionals in o_2 plus 1.

merge|numLoops. Given input functions i_1 and i_2, input function i_3 is created by merging i_1 and i_2. Assuming that o_1, o_2, and o_3 are their corresponding output functions, the number of loops in o_3 should be equal to the number of loops in o_1 plus the number of loops in o_2.

merge|compiles. Given input functions i_1 and i_2, input function i_3 is created by merging i_1 and i_2. Assuming that o_1, o_2, and o_3 are their corresponding output functions, if o_1 and o_2 compile, then o_3 should also compile.

merge|retValues. Given input functions i_1 and i_2, input function i_3 is created by merging i_1 and i_2. Assuming that o_1, o_2, and o_3 are their corresponding output functions, if o_1, o_2, and o_3 compile, then the return values of o_3 should be equal either to the return values of o_1 or o_2.

B.3 Specifications for Search

For our search procedure, we only specify 2-safety properties. We again name our properties based on the program-transformation and program-inspection functions they use and add an "S" at the end to differentiate them from the properties written for testing: <program-transformation function> | <program-inspection function> | S.

renameParam|arity|S. Given input function i_1, input function i_2 is created by renaming a random parameter in i_1. Assuming that o_1 and o_2 are their corresponding output functions, if the arity of o_2 is equal to the arity of i_2, then the arity of o_1 should also be equal to the arity of i_1.

renameParam|numConditionals|S. Given input function i_1, input function i_2 is created by renaming a random parameter in i_1. Assuming that o_1 and o_2 are their corresponding output functions, if the number of conditionals in o_2 is equal to the number of conditionals in i_2, then the number of conditionals in o_1 should also be equal to the number of conditionals in i_1.

renameParam|numLoops|S. Given input function i_1, input function i_2 is created by renaming a random parameter in i_1. Assuming that o_1 and o_2 are their corresponding output functions, if the number of loops in o_2 is equal to the number of loops in i_2, then the number of loops in o_1 should also be equal to the number of loops in i_1.

renameParam|compiles|S. Given input function i_1, input function i_2 is created by renaming a random parameter in i_1. Assuming that o_1 and o_2 are their corresponding output functions, if o_2 compiles, then o_1 should also compile.

renameParam|retValues|S. Given input function i_1, input function i_2 is created by renaming a random parameter in i_1. Assuming that o_1 and o_2 are their corresponding, *compiling* output functions, if the return values of o_2 are equal to the return values of i_2, then the return values of o_1 should also be equal to the return values of i_1.

addParam|arity|S. Given input function i_1, input function i_2 is created by adding a random parameter in i_1. Assuming that o_1 and o_2 are their corresponding output functions, if the arity of o_2 is equal to the arity of i_2, then the arity of o_1 should also be equal to the arity of i_1.

addParam|numConditionals|S. Given input function i_1, input function i_2 is created by adding a random parameter in i_1. Assuming that o_1 and o_2 are their corresponding output functions, if the number of conditionals in o_2 is equal to the number of conditionals in i_2, then the number of conditionals in o_1 should also be equal to the number of conditionals in i_1.

addParam|numLoops|S. Given input function i_1, input function i_2 is created by adding a random parameter in i_1. Assuming that o_1 and o_2 are their corresponding output functions, if the number of loops in o_2 is equal to the number of loops in i_2, then the number of loops in o_1 should also be equal to the number of loops in i_1.

addParam|compiles|S. Given input function i_1, input function i_2 is created by adding a random parameter in i_1. Assuming that o_1 and o_2 are their corresponding output functions, if o_2 compiles, then o_1 should also compile.

addParam|retValues|S. Given input function i_1, input function i_2 is created by adding a random parameter in i_1. Assuming that o_1 and o_2 are their corresponding, *compiling* output functions, if the return values of o_2 are equal to the return values of i_2, then the return values of o_1 should also be equal to the return values of i_1.

addConditional|arity|S. Given input function i_1, input function i_2 is created by adding a random conditional in i_1. Assuming that o_1 and o_2 are their corresponding output functions, if the arity of o_2 is equal to the arity of i_2, then the arity of o_1 should also be equal to the arity of i_1.

addConditional|numConditionals|S. Given input function i_1, input function i_2 is created by adding a random conditional in i_1. Assuming that o_1 and o_2 are their corresponding output functions, if the number of conditionals in o_2 is equal to the number of conditionals in i_2, then the number of conditionals in o_1 should also be equal to the number of conditionals in i_1.

addConditional|numLoops|S. Given input function i_1, input function i_2 is created by adding a random conditional in i_1. Assuming that o_1 and o_2 are their corresponding output functions, if the number of loops in o_2 is equal to the number of loops in i_2, then the number of loops in o_1 should also be equal to the number of loops in i_1.

addConditional|compiles|S. Given input function i_1, input function i_2 is created by adding a random conditional in i_1. Assuming that o_1 and o_2 are their corresponding output functions, if o_2 compiles, then o_1 should also compile.

addConditional|retValues|S. Given input function i_1, input function i_2 is created by adding a random conditional in i_1. Assuming that o_1 and o_2 are their corresponding, *compiling* output functions, if the return values of o_2 are equal to the return values of i_2, then the return values of o_1 should also be equal to the return values of i_1.

chBranchCond|arity|S. Given input function i_1, input function i_2 is created by replacing the condition of a random if- or switch-statement in i_1 with a random condition. Assuming that o_1 and o_2 are their corresponding output functions, if the arity of o_2 is equal to the arity of i_2, then the arity of o_1 should also be equal to the arity of i_1.

chBranchCond|numConditionals|S. Given input function i_1, input function i_2 is created by replacing the condition of a random if- or switch-statement in i_1 with a random condition. Assuming that o_1 and o_2 are their corresponding output functions, if the number of conditionals in o_2 is equal to the number of conditionals in i_2, then the number of conditionals in o_1 should also be equal to the number of conditionals in i_1.

chBranchCond|numLoops|S. Given input function i_1, input function i_2 is created by replacing the condition of a random if- or switch-statement in i_1 with a random condition. Assuming that o_1 and o_2 are their corresponding output functions, if the number of loops in o_2 is equal to the number of loops in i_2, then the number of loops in o_1 should also be equal to the number of loops in i_1.

chBranchCond|compiles|S. Given input function i_1, input function i_2 is created by replacing the condition of a random if- or switch-statement in i_1 with a random condition. Assuming that o_1 and o_2 are their corresponding output functions, if o_2 compiles, then o_1 should also compile.

addLoop|arity|S. Given input function i_1, input function i_2 is created by adding a random for-loop in i_1. Assuming that o_1 and o_2 are their corresponding output functions, if the arity of o_2 is equal to the arity of i_2, then the arity of o_1 should also be equal to the arity of i_1.

addLoop|numConditionals|S. Given input function i_1, input function i_2 is created by

adding a random for-loop in i_1. Assuming that o_1 and o_2 are their corresponding output functions, if the number of conditionals in o_2 is equal to the number of conditionals in i_2, then the number of conditionals in o_1 should also be equal to the number of conditionals in i_1.

addLoop|numLoops|S. Given input function i_1, input function i_2 is created by adding a random for-loop in i_1. Assuming that o_1 and o_2 are their corresponding output functions, if the number of loops in o_2 is equal to the number of loops in i_2, then the number of loops in o_1 should also be equal to the number of loops in i_1.

addLoop|compiles|S. Given input function i_1, input function i_2 is created by adding a random for-loop in i_1. Assuming that o_1 and o_2 are their corresponding output functions, if o_2 compiles, then o_1 should also compile.

addLoop|retValues|S. Given input function i_1, input function i_2 is created by adding a random for-loop in i_1. Assuming that o_1 and o_2 are their corresponding, *compiling* output functions, if the return values of o_2 are equal to the return values of i_2, then the return values of o_1 should also be equal to the return values of i_1.

B.4 Further Experimental Results

Tables B.1, B.2, and B.3 present our additional experimental results for Chapter 4.

Table B.1: The number of unique property violations detected when running the testing procedure on TRANSCODER and TRANSCODER-IR for translating from Java to C++. The first column (BS) shows the beam-size parameter, and the second column the value of k of the corresponding k-safety property. The third column shows the (abbreviated) program-transformation (PT) functions, which are combined with the (abbreviated) program-inspection (PI) functions of the first row to form the properties in Appendix B.2. We report the number of violations under each PI function—the left sub-column shows the violations for TRANSCODER and the right for TRANSCODER-IR. For example, the violations of the property in Figure 4.1b are shown in the k=1 rows under `retV/s`: for BS=1, there are 254 violations for TRANSCODER and 234 for TRANSCODER-IR.

BS	k	PI / PT	`arity`		`numC/s`		`numL/s`		`compiles`		`retV/s`	
1	1	–	0	0	1	1	0	2	96	109	254	234
	2	`rnmP`	0	1	3	3	0	7	351	232	209	474
		`addP`	0	0	4	11	3	19	843	902	489	591
		`addC`	0	0	4	1	0	11	1006	416	1119	652
		`chBC`	0	0	15	8	9	10	1305	1355	–	–
		`addL`	0	0	8	9	2	3	920	692	371	974
		`rmL`	0	0	0	0	3	0	49	261	–	–
	3	`mrg`	5	8	17	9	11	23	1514	1525	52	60
3	1	–	0	0	0	0	0	1	25	30	113	122
	2	`rnmP`	0	0	1	1	0	0	1	109	0	8
		`addP`	0	0	0	6	0	1	7	44	219	174
		`addC`	0	0	0	4	0	0	14	41	189	206
		`chBC`	0	0	13	13	9	10	98	427	–	–
		`addL`	0	0	5	5	1	0	7	29	2	457
		`rmL`	0	0	0	0	0	0	6	13	–	–
	3	`mrg`	4	3	10	10	6	7	305	594	0	11

Table B.2: The number of unique property violations detected when running the testing procedure on TRANSCODER, DOBF, and STARCODER for translating from Java to Python. Under each PI function, the left sub-column shows the violations for TRANSCODER, the middle for DOBF, and the right for STARCODER. For example, the violations of the property in Figure 4.1d are shown in the k=2, `rnmP` rows under `retV/s`: for BS=1, there are 131 violations for TRANSCODER, 147 for DOBF, and 437 for STARCODER.

BS	k	PI / PT	arity			numC/s			numL/s			compiles			retV/s		
1	1	–	0	0	1	8	2	14	9	17	13	103	84	32	211	237	294
	2	rnmP	0	0	0	36	32	77	46	23	961	188	177	101	131	147	437
		addP	0	0	18	50	32	88	76	34	90	176	204	1168	131	117	351
		addC	0	1	3	28	13	2162	58	36	78	131	216	299	626	428	206
		chBC	0	0	18	51	21	467	81	28	102	1086	1184	466	–	–	–
		addL	0	0	1	21	32	134	43	14	1053	209	208	286	621	650	629
		rmL	0	0	2	3	0	80	33	14	99	109	113	64	–	–	–
	3	mrg	0	2	8	166	94	244	220	108	175	994	1134	480	132	162	512
3	1	–	0	0	0	4	0	9	2	12	4	80	55	25	136	173	288
	2	rnmP	0	0	0	1	1	13	3	3	18	11	5	19	3	6	34
		addP	0	0	11	0	2	7	5	5	19	12	10	77	7	1	54
		addC	0	0	0	0	0	2038	7	4	25	12	21	191	382	212	75
		chBC	0	0	0	16	8	279	7	5	26	706	878	382	–	–	–
		addL	0	0	0	1	1	21	10	2	565	23	24	89	338	429	469
		rmL	0	0	0	0	0	7	1	5	78	50	38	33	–	–	–
	3	mrg	0	0	0	19	11	82	30	16	62	368	546	1391	6	62	220

Table B.3: The average running time (in seconds), including model inference and test harness execution, for checking a property on each model.

BS	Average Running Time (s)				
	TRANSCODER Java-C++	TRANSCODER-IR Java-C++	TRANSCODER Java-Python	DOBF Java-Python	STARCODER Java-Python
1	2.0	2.1	1.5	1.3	42.9
3	5.9	6.8	2.8	3.3	51.3

List of Tables

List of Figures

List of Algorithms

Bibliography

(Last accessed: June 17, 2024). Libfuzzer—A library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

(Last accessed: June 17, 2024). Technical "whitepaper" for AFL. http://lcamtuf.coredump.cx/afl/technical_details.txt.

Akazaki, T., Liu, S., Yamagata, Y., Duan, Y., and Hao, J. (2018). Falsification of cyber-physical systems using deep reinforcement learning. In *FM*, volume 10951 of *LNCS*, pages 456–465. Springer.

Albarghouthi, A., D'Antoni, L., Drews, S., and Nori, A. V. (2017). FairSquare: Probabilistic verification of program fairness. *PACMPL*, **1**, 80:1–80:30.

Alshiekh, M., Bloem, R., Ehlers, R., Könighofer, B., Niekum, S., and Topcu, U. (2018). Safe reinforcement learning via shielding. In *AAAI*, pages 2669–2678. AAAI.

Amazon-Q (2024). Amazon q developer. *Amazon*.

Amazon-Robotics (2022). Amazon robotics: Robots in fulfillment centers. *Amazon*.

Athiwaratkun, B., Gouda, S. K., Wang, Z., Li, X., Tian, Y., Tan, M., Ahmad, W. U., Wang, S., Sun, Q., Shang, M., Gonugondla, S. K., Ding, H., Kumar, V., Fulton, N., Farahani, A., Jain, S., Giaquinto, R., Qian, H., Ramanathan, M. K., and Nallapati, R. (2023). Multi-lingual evaluation of code generation models. In *ICLR*. OpenReview.net.

Austin, J., Odena, A., Nye, M. I., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C. J., Terry, M., Le, Q. V., and Sutton, C. (2021). Program synthesis with large language models. *CoRR*, **abs/2108.07732**.

Banerjee, D., Xu, C., and Singh, G. (2024). Input-relational verification of deep neural networks. *Proc. ACM Program. Lang.*, **8**(PLDI), 1–27.

Bastani, O., Pu, Y., and Solar-Lezama, A. (2018). Verifiable reinforcement learning via policy extraction. In *NeurIPS*, pages 2499–2509.

Bastani, O., Zhang, X., and Solar-Lezama, A. (2019). Probabilistic verification of fairness properties via concentration. *PACMPL*, **3**, 118:1–118:27.

Berrada, L., Dathathri, S., Dvijotham, K., Stanforth, R., Bunel, R., Uesato, J., Gowal, S., and Kumar, M. P. (2021). Make sure you're unsure: A framework for verifying probabilistic specifications. In *NeurIPS*, pages 11136–11147.

Beurer-Kellner, L., Fischer, M., and Vechev, M. T. (2023). Prompting is programming: A query language for large language models. In *PLDI*. ACM. To appear.

Biagiola, M. and Tonella, P. (2024). Testing of deep reinforcement learning agents with surrogate models. *Trans. Softw. Eng. Methodol.*, **33**(3), 73:1–73:33.

Bonet, B. and Geffner, H. (2001). Planning as heuristic search. *AIJ*, **129**, 5–33.

Boston-Dynamics (2024). Boston dynamics. *Boston Dynamics*.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). OpenAI Gym. *CoRR*, **abs/1606.01540**.

Carlini, N. and Wagner, D. A. (2017). Towards evaluating the robustness of neural networks. In *S&P*, pages 39–57. IEEE Computer Society.

Carr, S., Jansen, N., Junges, S., and Topcu, U. (2023). Safe reinforcement learning via shielding under partial observability. In *AAAI*, pages 14748–14756. AAAI Press.

Cassano, F., Gouwar, J., Nguyen, D., Nguyen, S., Phipps-Costin, L., Pinckney, D., Yee, M., Zi, Y., Anderson, C. J., Feldman, M. Q., Guha, A., Greenberg, M., and Jangda, A. (2023). MultiPL-E: A scalable and polyglot approach to benchmarking neural code generation. *TSE*, **49**, 3675–3691.

Cer, D., Yang, Y., Kong, S., Hua, N., Limtiaco, N., St. John, R., Constant, N., Guajardo-Cespedes, M., Yuan, S., Tar, C., Sung, Y., Strope, B., and Kurzweil, R. (2018). Universal sentence encoder. *CoRR*, **abs/1803.11175**.

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. (2021). Evaluating large language models trained on code. *CoRR*, **abs/2107.03374**.

Chen, T. Y., Cheung, S. C., and Yiu, S. (1998). Metamorphic testing: A new approach for generating next test cases. Technical Report HKUST–CS98–01, HKUST.

Christakis, M., Eniser, H. F., Hoffmann, J., Singla, A., and Wüstholz, V. (2023). Specifying and testing k-safety properties for machine-learning models. In *IJCAI*.

Clarkson, M. R. and Schneider, F. B. (2008). Hyperproperties. In *CSF*, pages 51–65. IEEE Computer Society.

Continue.dev (2024). Continue.dev. *Continue.dev*.

Corso, A., Moss, R. J., Koren, M., Lee, R., and Kochenderfer, M. J. (2021). A survey of algorithms for black-box safety validation. *JAIR*, **72**, 377–428.

Cui, J., Liew, L. S., Sabaliauskaite, G., and Zhou, F. (2019). A review on safety failures, security attacks, and available countermeasures for autonomous vehicles. *Ad Hoc Networks*, **90**, 101823. Recent advances on security and privacy in Intelligent Transportation Systems.

Cursor (2024). Cursor. *Cursor*.

Dalrymple, D., Skalse, J., Bengio, Y., Russell, S., Tegmark, M., Seshia, S., Omohundro, S., Szegedy, C., Goldhaber, B., Ammann, N., Abate, A., Halpern, J., Barrett, C., Zhao, D., Zhi-Xuan, T., Wing, J., and Tenenbaum, J. (2024). Towards guaranteed safe ai: A framework for ensuring robust and reliable ai systems. *CoRR*.

Deng, Y., Zheng, X., Zhang, T., Lou, G., Liu, H., and Kim, M. (2020). RMT: Rule-based metamorphic testing for autonomous driving models. *CoRR*, **abs/2012.10672**.

Deng, Y., Lou, G., Zheng, J. X., Zhang, T., Kim, M., Liu, H., Wang, C., and Chen, T. Y. (2021). BMT: Behavior driven development-based metamorphic testing for autonomous driving models. In *MET@ICSE*, pages 32–36. IEEE Computer Society.

Deng, Y., Zheng, X., Zhang, T., Liu, H., Lou, G., Kim, M., and Chen, T. Y. (2022). A declarative metamorphic testing framework for autonomous driving. *TSE*, pages 1–20.

Domshlak, C., Hoffmann, J., and Katz, M. (2015). Red-black planning: A new systematic approach to partial delete relaxation. *AIJ*, **221**, 73–114.

Dreossi, T., Dang, T., Donzé, A., Kapinski, J., Jin, X., and Deshmukh, J. V. (2015). Efficient guiding strategies for testing of temporal properties of hybrid systems. In *NFM*, volume 9058 of *LNCS*, pages 127–142. Springer.

Dwarakanath, A., Ahuja, M., Sikand, S., Rao, R. M., Bose, R. P. J. C., Dubash, N., and Podder, S. (2018). Identifying implementation bugs in machine learning based image classifiers using metamorphic testing. In *ISSTA*, pages 118–128. ACM.

Edelkamp, S. (2001). Planning with pattern databases. In *ECP*, pages 13–24. AAAI.

Eisenhut, J., Torralba, Á., Christakis, M., and Hoffmann, J. (2023). Automatic metamorphic test oracles for action-policy testing. In *ICAPS*, pages 109–117.

Eisenhut, J., Schuler, X., Fiser, D., Höller, D., Christakis, M., and Hoffmann, J. (2024). New fuzzing biases for action policy testing. In *ICAPS*, pages 162–167.

Eniser, H. F., Gros, T. P., Wüstholz, V., Hoffmann, J., and Christakis, M. (2022). Metamorphic relations via relaxations: An approach to obtain oracles for action-policy testing. In *ISSTA*, pages 52–63. ACM.

Eniser, H. F., Wüstholz, V., and Christakis, M. (2024a). Automatically testing functional properties of code translation models. In *AAAI*, pages 21055–21062.

Eniser, H. F., Zhang, H., David, C., Wang, M., Christakis, M., Paulsen, B., Dodds, J., and Kroening, D. (2024b). Towards translating real-world code with llms: A study of translating to rust. *CoRR*, **abs/2405.11514**.

Ernst, G., Sedwards, S., Zhang, Z., and Hasuo, I. (2019). Fast falsification of hybrid systems using probabilistically adaptive input. In *QEST*, volume 11785 of *LNCS*, pages 165–181. Springer.

Fluri, L., Paleka, D., and Tramèr, F. (2024). Evaluating superhuman models with consistency checks. In *SaTML*, pages 194–232. IEEE.

Galhotra, S., Brun, Y., and Meliou, A. (2017). Fairness testing: Testing software for discrimination. In *ESEC/FSE*, pages 498–510. ACM.

García, J. and Fernández, F. (2015). A comprehensive survey on safe reinforcement learning. *JMLR*, **16**, 1437–1480.

Garg, S., Bajpai, A., and Mausam (2019). Size independent neural transfer for RDDL planning. In *ICAPS*, pages 631–636. AAAI.

Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., and Vechev, M. T. (2018). AI2: Safety and robustness certification of neural networks with abstract interpretation. In *S&P*, pages 3–18. IEEE Computer Society.

Gentilini, R., Piazza, C., and Policriti, A. (2003). From bisimulation to simulation: Coarsest partition problems. *J. Autom. Reason.*, **31**, 73–103.

Gerasimou, S., Eniser, H. F., Sen, A., and Çakan, A. (2020). Importance-driven deep learning system testing. In *ICSE*, pages 322–323. ACM.

Github-Copilot (2024). Github copilot. *Github*.

Godefroid, P. (2020). Fuzzing: Hack, art, and science. *CACM*, **63**, 70–76.

Goodfellow, I. J., Shlens, J., and Szegedy, C. (2015). Explaining and harnessing adversarial examples. In *ICLR*.

Groshev, E., Goldstein, M., Tamar, A., Srivastava, S., and Abbeel, P. (2018). Learning generalized reactive policies using deep neural networks. In *ICAPS*, pages 408–416. AAAI.

Gu, A., Rozière, B., Leather, H. J., Solar-Lezama, A., Synnaeve, G., and Wang, S. (2024). Cruxeval: A benchmark for code reasoning, understanding and execution. In *ICML*. OpenReview.net.

Haarnoja, T., Moran, B., Lever, G., Huang, S. H., Tirumala, D., Humplik, J., Wulfmeier, M., Tunyasuvunakool, S., Siegel, N. Y., Hafner, R., Bloesch, M., Hartikainen, K., Byravan, A., Hasenclever, L., Tassa, Y., Sadeghi, F., Batchelor, N., Casarini, F., Saliceti, S., Game, C., Sreendra, N., Patel, K., Gwira, M., Huber, A., Hurley, N., Nori, F., Hadsell, R., and Heess, N. (2024). Learning agile soccer skills for a bipedal robot with deep reinforcement learning. *Science Robotics*, **9**(89).

He, J., Yang, Z., Shi, J., Yang, C., Kim, K., Xu, B., Zhou, X., and Lo, D. (2024). Curiosity-driven testing for sequential decision-making process. In *ICSE*, pages 165:1–165:14.

He, P., Meister, C., and Su, Z. (2020). Structure-invariant testing for machine translation. In *ICSE*, pages 961–973. ACM.

Helmert, M., Haslum, P., Hoffmann, J., and Nissim, R. (2014). Merge-and-shrink abstraction: A method for generating lower bounds in factored state spaces. *JACM*, **61**, 16:1–16:63.

Ho, J. and Ermon, S. (2016). Generative adversarial imitation learning. In *NeurIPS*, pages 4565–4573.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *CACM*, **12**, 576–580.

Hofmann, H. (1994). The German credit dataset. https://archive.ics.uci.edu/ml/datasets/Statlog+(German+Credit+Data).

Huang, X., Kwiatkowska, M., Wang, S., and Wu, M. (2017). Safety verification of deep neural networks. In *CAV*, volume 10426 of *LNCS*, pages 3–29. Springer.

Hunt, N., Fulton, N., Magliacane, S., Hoang, T. N., Das, S., and Solar-Lezama, A. (2021). Verifiably safe exploration for end-to-end reinforcement learning. In *HSCC*, pages 14:1–14:11. ACM.

Ibrahimzada, A. R., Ke, K., Pawagi, M., Abid, M. S., Pan, R., Sinha, S., and Jabbarvand, R. (2024). Repository-level compositional code translation and validation. *CoRR*, **abs/2410.24117**.

Issakkimuthu, M., Fern, A., and Tadepalli, P. (2018). Training deep reactive policies for probabilistic planning problems. In *ICAPS*, pages 422–430. AAAI.

Jamsaz, A. T., Bhattacharjee, A., Chen, L., Ahmed, N. K., Yazdanbakhsh, A., and Jannesari, A. (2024). Coderosetta: Pushing the boundaries of unsupervised code translation for parallel programming. In *NeurIPS*.

Karia, R. and Srivastava, S. (2021). Learning generalized relational heuristic networks for model-agnostic planning. In *AAAI*, pages 8064–8073. AAAI.

Katz, G., Barrett, C. W., Dill, D. L., Julian, K., and Kochenderfer, M. J. (2017). Reluplex: An efficient SMT solver for verifying deep neural networks. In *CAV*, volume 10426 of *LNCS*, pages 97–117. Springer.

Khedr, H. and Shoukry, Y. (2023). Certifair: A framework for certified global fairness of neural networks. In *AAAI*, pages 8237–8245.

Khlaaf, H. (2023). Toward comprehensive risk assessments and assurance of ai-based systems. *Trail of Bits*.

Kim, J., Feldt, R., and Yoo, S. (2019). Guiding deep learning system testing using surprise adequacy. In *ICSE*, pages 1039–1049. IEEE Computer Society/ACM.

Könighofer, B., Lorber, F., Jansen, N., and Bloem, R. (2020). Shield synthesis for reinforcement learning. In *ISoLA*, pages 290–306. Springer.

Könighofer, B., Rudolf, J., Palmisano, A., Tappler, M., and Bloem, R. (2023). Online shielding for reinforcement learning. *Innov. Syst. Softw. Eng.*, **19**(4), 379–394.

Koren, M., Alsaif, S., Lee, R., and Kochenderfer, M. J. (2018). Adaptive stress testing for autonomous vehicles. In *IV*, pages 1–7. IEEE Computer Society.

Lachaux, M., Rozière, B., Szafraniec, M., and Lample, G. (2021). DOBF: A deobfuscation pre-training objective for programming languages. In *NeurIPS*, pages 14967–14979.

Larson, J., Mattu, S., Kirchner, L., and Angwin, J. (2016). How we analyzed the COMPAS recidivism algorithm. https://www.propublica.org/article/how-we-analyzed-the-compas-recidivism-algorithm.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. In *Proc. IEEE*, pages 2278–2324. IEEE Computer Society.

LeCun, Y., Cortes, C., and Burges, C. J. (1999). The MNIST database of handwritten digits. http://yann.lecun.com/exdb/mnist.

Lee, R., Mengshoel, O. J., Saksena, A., Gardner, R. W., Genin, D., Silbermann, J., Owen, M. P., and Kochenderfer, M. J. (2020). Adaptive stress testing: Finding likely failure events with reinforcement learning. *JAIR*, **69**, 1165–1201.

Leuschel, M. and Butler, M. J. (2003). Prob: A model checker for B. In *FME*, volume 2805, pages 855–874.

Li, R., Li, J., Huang, C., Yang, P., Huang, X., Zhang, L., Xue, B., and Hermanns, H. (2020). PRODeep: A platform for robustness verification of deep neural networks. In *ESEC/FSE*, pages 1630–1634. ACM.

Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., Liu, Q., Zheltonozhskii, E., Zhuo, T. Y., Wang, T., Dehaene, O., Davaadorj, M., Lamy-Poirier, J., Monteiro, J., Shliazhko, O., Gontier, N., Meade, N., Zebaze, A., Yee, M., Umapathi, L. K., Zhu, J., Lipkin, B., Oblokulov, M., Wang, Z., V, R. M., Stillerman, J., Patel, S. S., Abulkhanov, D., Zocca, M., Dey, M., Zhang, Z., Moustafa-Fahmy, N., Bhattacharyya, U., Yu, W., Singh, S., Luccioni, S., Villegas, P., Kunakov, M., Zhdanov, F., Romero, M., Lee, T., Timor, N., Ding, J., Schlesinger, C., Schoelkopf, H., Ebert, J., Dao, T., Mishra, M., Gu, A., Robinson, J., Anderson, C. J., Dolan-Gavitt, B., Contractor, D., Reddy, S., Fried, D., Bahdanau, D., Jernite, Y., Ferrandis, C. M., Hughes, S., Wolf, T., Guha, A., von Werra, L., and de Vries, H. (2023a). StarCoder: May the source be with you! *CoRR*, **abs/2305.06161**.

Li, X. L., Shrivastava, V., Li, S., Hashimoto, T., and Liang, P. (2024). Benchmarking and improving generator-validator consistency of language models. In *ICLR*.

Li, Z., Wu, X., Zhu, D., Cheng, M., Chen, S., Zhang, F., Xie, X., Ma, L., and Zhao, J. (2023b). Generative model-based testing on decision-making policies. In *ASE*, pages 243–254. IEEE.

Liu, J. (2017). 515K hotel reviews data in Europe. https://www.kaggle.com/datasets/jiashenliu/515k-hotel-reviews-data-in-europe.

Liu, J., Xia, C. S., Wang, Y., and Zhang, L. (2023). Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. *CoRR*, **abs/2305.01210**.

Luu, Q.-H., Liu, H., Chen, T. Y., and Vu, H. L. (2024). A sequential metamorphic testing framework for understanding autonomous vehicle's decisions. *IEEE Transactions on Intelligent Vehicles*, pages 1–13.

Ma, L., Juefei-Xu, F., Zhang, F., Sun, J., Xue, M., Li, B., Chen, C., Su, T., Li, L., Liu, Y., Zhao, J., and Wang, Y. (2018). DeepGauge: Multi-granularity testing criteria for deep learning systems. In *ASE*, pages 120–131. ACM.

Ma, P., Wang, S., and Liu, J. (2020). Metamorphic testing and certified mitigation of fairness violations in NLP models. In *IJCAI*, pages 458–465. ijcai.org.

Macedo, M., Tian, Y., Nie, P., Côgo, F. R., and Adams, B. (2024). Intertrans: Leveraging transitive intermediate translations to enhance llm-based code translation. *CoRR*, **abs/2411.01063**.

Macrae, C. (2022). Learning from the failure of autonomous and intelligent systems: Accidents, safety, and sociotechnical sources of risk. *Risk Analysis*, **42**(9), 1999–2025.

Madry, A., Makelov, A., Schmidt, L., Tsipras, D., and Vladu, A. (2018). Towards deep learning models resistant to adversarial attacks. In *ICLR*. OpenReview.net.

Mazouni, Q., Spieker, H., Gotlieb, A., and Acher, M. (2024). Testing for fault diversity in reinforcement learning. In *AST*, pages 136–146. ACM.

Mazouni, Q., Gotlieb, A., Spieker, H., Acher, M., and Combemale, B. (2025). Mutation-guided metamorphic testing of optimality in ai planning. *Software Testing, Verification and Reliability*, **35**(1).

Meyer, B. (1992). *Eiffel: The Language*. Prentice-Hall.

Miller, B. P., Fredriksen, L., and So, B. (1990). An empirical study of the reliability of UNIX utilities. *CACM*, **33**, 32–44.

Milner, R. (1971). An algebraic definition of simulation between programs. In *IJCAI*, pages 481–489.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M. A., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, **518**, 529–533.

Odena, A., Olsson, C., Andersen, D., and Goodfellow, I. J. (2019). TensorFuzz: Debugging neural networks with coverage-guided fuzzing. In *ICML*, volume 97 of *PMLR*, pages 4901–4911. PMLR.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P. F., Leike, J., and Lowe, R. (2022). Training language models to follow instructions with human feedback. In *NeurIPS*.

Pan, R., Ibrahimzada, A. R., Krishna, R., Sankar, D., Wassi, L. P., Merler, M., Sobolev, B., Pavuluri, R., Sinha, S., and Jabbarvand, R. (2024). Lost in translation: A study of bugs introduced by large language models while translating code. In *ICSE*, pages 82:1–82:13. ACM.

Pang, Q., Yuan, Y., and Wang, S. (2021). MDPFuzzer: Finding crash-triggering state sequences in models solving the Markov decision process. *CoRR*, **abs/2112.02807**.

Parr, T. (2013). *The Definitive ANTLR 4 Reference, 2nd Edition*. O'Reilly Media.

Pei, K., Cao, Y., Yang, J., and Jana, S. (2017). DeepXplore: Automated whitebox testing of deep learning systems. In *SOSP*, pages 1–18. ACM.

Poesia, G., Polozov, A., Le, V., Tiwari, A., Soares, G., Meek, C., and Gulwani, S. (2022). Synchromesh: Reliable code generation from pre-trained language models. In *ICLR*. OpenReview.net.

Raffin, A., Hill, A., Ernestus, M., Gleave, A., Kanervisto, A., and Dormann, N. (2019). Stable baselines3. https://github.com/DLR-RM/stable-baselines3.

Rozière, B., Lachaux, M., Chanussot, L., and Lample, G. (2020). Unsupervised translation of programming languages. In *NeurIPS*.

Rozière, B., Zhang, J., Charton, F., Harman, M., Synnaeve, G., and Lample, G. (2022). Leveraging automated unit tests for unsupervised code translation. In *ICLR*. OpenReview.net.

Saha, S. K., Rabbi, F., Wang, S., and Yang, J. (2024). Specification-driven code translation powered by large language models: How far are we?

Savitzky, A. and Golay, M. J. E. (1964). Smoothing and differentiation of data by simplified least squares procedures. *Analytical Chemistry*, **36**, 1627–1639.

Scholak, T., Schucher, N., and Bahdanau, D. (2021). PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. In *EMNLP*, pages 9895–9901. ACM.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *CoRR*, **abs/1707.06347**.

Schultz, J., Adamek, J., Jusup, M., Lanctot, M., Kaisers, M., Perrin, S., Hennes, D., Shar, J., Lewis, C., Ruoss, A., Zahavy, T., Veličković, P., Prince, L., Singh, S., Malmi, E., and Tomašev, N. (2024). Mastering board games by external and internal planning with language models.

Segura, S., Fraser, G., Sánchez, A. B., and Cortés, A. R. (2016). A survey on metamorphic testing. *TSE*, **42**, 805–824.

Seshia, S. A., Desai, A., Dreossi, T., Fremont, D. J., Ghosh, S., Kim, E., Shivakumar, S., Vazquez-Chanlatte, M., and Yue, X. (2018). Formal specification for deep neural networks. In *ATVA*, volume 11138 of *LNCS*, pages 20–34. Springer.

Sharma, A. and Wehrheim, H. (2020). Higher income, larger loan? Monotonicity testing of machine learning models. In *ISSTA*, pages 200–210. ACM.

Shetty, M., Jain, N., Godbole, A., Seshia, S. A., and Sen, K. (2024). Syzygy: Dual code-test c to (safe) rust translation using llms and dynamic analysis.

Shin, R., Lin, C. H., Thomson, S., Chen, C., Roy, S., Platanios, E. A., Pauls, A., Klein, D., Eisner, J., and Durme, B. V. (2021). Constrained language models yield few-shot semantic parsers. In *EMNLP*, pages 7699–7715. ACM.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T. P., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, **529**, 484–489.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., and Hassabis, D. (2018). A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, **362**, 1140–1144.

Singh, G., Gehr, T., Püschel, M., and Vechev, M. T. (2019). An abstract domain for certifying neural networks. *PACMPL*, **3**, 41:1–41:30.

Sousa, M. and Dillig, I. (2016). Cartesian Hoare logic for verifying k-safety properties. In *PLDI*, pages 57–69. ACM.

Ståhlberg, S., Bonet, B., and Geffner, H. (2022). Learning general optimal policies with graph neural networks: Expressive power, transparency, and limits. In *ICAPS*, pages 629–637.

Steinmetz, M., Gros, T. P., Heim, P., Höller, D., and Hoffmann, J. (2021). Debugging a policy: A framework for automatic action policy testing. In *PRL@ICAPS*.

Sun, Y., Wu, M., Ruan, W., Huang, X., Kwiatkowska, M., and Kroening, D. (2018). Concolic testing for deep neural networks. In *ASE*, pages 109–119. ACM.

Sun, Z., Zhang, J. M., Xiong, Y., Harman, M., Papadakis, M., and Zhang, L. (2022). Improving machine translation systems via isotopic replacement. In *ICSE*, pages 1181–1192. ACM.

Szafraniec, M., Rozière, B., Leather, H., Labatut, P., Charton, F., and Synnaeve, G. (2023). Code translation with compiler representations. In *ICLR*. OpenReview.net.

Tang, Z., Agarwal, M., Shypula, A., Wang, B., Wijaya, D., Chen, J., and Kim, Y. (2023). Explain-then-translate: an analysis on improving program translation with self-generated explanations. In *EMNLP*, pages 1741–1788. Association for Computational Linguistics.

Tappler, M., Cano Córdoba, F., Aichernig, B. K., and Könighofer, B. (2022). Search-based testing of reinforcement learning. In *IJCAI*, pages 503–510.

Tian, Y., Pei, K., Jana, S., and Ray, B. (2018). DeepTest: Automated testing of deep-neural-network-driven autonomous cars. In *ICSE*, pages 303–314. ACM.

Tian, Y., Zhong, Z., Ordonez, V., Kaiser, G. E., and Ray, B. (2020). Testing DNN image classifiers for confusion & bias errors. In *ICSE*, pages 1122–1134. ACM.

Towers, M., Kwiatkowski, A., Terry, J. K., Balis, J. U., de Cola, G., Deleu, T., Goulão, M., Kallinteris, A., Krimmel, M., KG, A., Perez-Vicente, R., Pierré, A., Schulhoff, S., Tai, J. J., Tan, H. J. S., and Younis, O. G. (2024). Gymnasium: A Standard Interface for Reinforcement Learning Environments.

Toyer, S., Thiébaux, S., Trevizan, F. W., and Xie, L. (2020). ASNets: Deep learning for generalised planning. *JAIR*, **68**, 1–68.

Tramèr, F., Atlidakis, V., Geambasu, R., Hsu, D. J., Hubaux, J., Humbert, M., Juels, A., and Lin, H. (2017). FairTest: Discovering unwarranted associations in data-driven applications. In *EuroS&P*, pages 401–416. IEEE Computer Society.

Udeshi, S., Arora, P., and Chattopadhyay, S. (2018). Automated directed fairness testing. In *ASE*, pages 98–108. ACM.

Urban, C., Christakis, M., Wüstholz, V., and Zhang, F. (2020). Perfectly parallel fairness certification of neural networks. *PACMPL*, **4**, 185:1–185:30.

Usman, M., Noller, Y., Pasareanu, C. S., Sun, Y., and Gopinath, D. (2021). NEUROSPF: A tool for the symbolic analysis of neural networks. In *ICSE*, pages 25–28. IEEE Computer Society.

Varshosaz, M., Ghaffari, M., Johnsen, E. B., and Wasowski, A. (2023). Formal specification and testing for reinforcement learning. *Program. Lang.*, **7**(ICFP), 125–158.

Vu, F., Dunkelau, J., and Leuschel, M. (2024). Validation of reinforcement learning agents and safety shields with prob. In *NASA Formal Methods (NFM) 2024*, volume 14627, pages 279–297. Springer.

Wang, J., Dong, G., Sun, J., Wang, X., and Zhang, P. (2019). Adversarial sample detection for deep neural network through model mutation testing. In *ICSE*, pages 1245–1256. IEEE Computer Society/ACM.

Wang, S. and Su, Z. (2020). Metamorphic object insertion for testing object detection systems. In *ASE*, pages 1053–1065. IEEE Computer Society.

Wang, S., Pei, K., Whitehouse, J., Yang, J., and Jana, S. (2018). Formal security analysis of neural networks using symbolic intervals. In *Security*, pages 1599–1614. USENIX.

Wang, S., Zhang, H., Xu, K., Lin, X., Jana, S., Hsieh, C., and Kolter, J. Z. (2021). Beta-CROWN: Efficient bound propagation with per-neuron split constraints for neural network robustness verification. In *NeurIPS*, pages 29909–29921.

Wang, S., Li, Z., Qian, H., Yang, C., Wang, Z., Shang, M., Kumar, V., Tan, S., Ray, B., Bhatia, P., Nallapati, R., Ramanathan, M. K., Roth, D., and Xiang, B. (2023). ReCode: Robustness evaluation of code generation models. In *ACL*, pages 13818–13843. ACL.

Wang, Z., Huang, C., and Zhu, Q. (2022). Efficient global robustness certification of neural networks via interleaving twin-network encoding. In *DATE*, pages 1087–1092. IEEE.

Warden, P. (2018). Speech commands: A dataset for limited-vocabulary speech recognition. *CoRR*, **abs/1804.03209**.

Wicker, M., Huang, X., and Kwiatkowska, M. (2018). Feature-guided black-box safety testing of deep neural networks. In *TACAS*, volume 10805 of *LNCS*, pages 408–426. Springer.

Yang, P., Li, R., Li, J., Huang, C., Wang, J., Sun, J., Xue, B., and Zhang, L. (2021). Improving neural network verification through spurious region guided refinement. In *TACAS*, volume 12651 of *LNCS*, page 12651. Springer.

Yang, W.-C., Marra, G., Rens, G., and De Raedt, L. (2023). Safe reinforcement learning via probabilistic logic shields. In *IJCAI*, pages 5739–5749.

Zeng, Y., Shi, Z., Jin, M., Kang, F., Lyu, L., Hsieh, C., and Jia, R. (2023). Towards robustness certification against universal perturbations. In *ICLR*.

Zhang, M., Zhang, Y., Zhang, L., Liu, C., and Khurshid, S. (2018). DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems. In *ASE*, pages 132–142. ACM.

Zhang, P., Wang, J., Sun, J., Dong, G., Wang, X., Wang, X., Dong, J. S., and Dai, T. (2020). White-box fairness testing through adversarial sampling. In *ICSE*, pages 949–960. ACM.

Zhou, Z. Q. and Sun, L. (2019). Metamorphic testing of driverless cars. *CACM*, **62**, 61–67.

Hasan Ferit Enişer

Contact: https://hfeniser.github.io

Research Imperative

My research focuses on making AI models and AI-enabled systems reliable. In technical terms, I specialize in the intersection of program analysis and AI. I have a background in fuzzing and formal verification. Additionally, I have practical experience in training and using various AI models, including action policies and Large Language Models.

Education

I completed my PhD at the *Max Planck Institute for Software Systems* in 2025 under the supervision of Dr. Maria Christakis. I completed my B.Sc. and M.Sc. studies at *Bogazici University* in 2015 and 2018, respectively.

Work Experience

Since October 2024, I have been working at Amazon as an Applied Scientist. Previously, I completed two internships at the same company. Before that, I worked at Bogazici University as a Teaching Assistant and at Yupana Inc. as a part-time software engineer.

Research Publications

Peer-Reviewed Journal Publications

[1] Virtualization of Stateful Services via Machine Learning
Software Quality Journal
Hasan F. Eniser[†], Alper Sen (2020)
Impact Factor: 2.1
Part of MSc Thesis

Peer-Reviewed Conference Publications

[1] Using Action-Policy Testing in RL to Reduce the Number of Bugs
Symposium on Combinatorial Search ***(SoCS)*** *2025*
Hasan F. Eniser[†], Songtuan Lin, Nicola Müller, Anastasia Isychev, Valentin Wüstholz, Isabel Valera, Jörg Hoffmann and Maria Christakis
CORE Rank: B **Acceptance Rate:** N/A

[2] Automatically Testing Functional Properties of Code Translation Models
Association for the Advancement of Artificial Intelligence ***(AAAI)*** *2024*
Hasan F. Eniser[†], Maria Christakis, Valentin Wuestholz
CORE Rank: A* **Acceptance Rate:** 18.4%

[3] Specifying and Testing k-Safety Properties for Machine-Learning Models
International Joint Conference on Artificial Intelligence ***(IJCAI)*** *2023*
Maria Christakis, *Hasan F. Eniser*[†], Joerg Hoffmann, Adish Singla, Valentin Wuestholz
CORE Rank: A* **Acceptance Rate:** ∼15%

[†]Lead author.

[4] Metamorphic Relations via Relaxations: An Approach to Obtain Oracles for Action-Policy Testing
International Symposium on Software Testing and Analysis ***(ISSTA)*** *2022*
Hasan F. Eniser[†], Timo Gros, Valentin Wuestholz, Joerg Hoffmann, Maria Christakis
CORE Rank: A **Acceptance Rate:** 24.4%

[5] Debugging a Policy: Automatic Action-Policy Testing in AI Planning
International Conference on Automated Planning and Scheduling ***(ICAPS)*** *2022*
Marcel Steinmetz, Daniel Fiser, *Hasan F. Eniser*, Patrick Ferber, Timo Gros, Philippe Heim, Daniel Hoeller, Xandra Schuler, Valentin Wuestholz, Maria Christakis and Joerg Hoffmann
CORE Rank: A* **Acceptance Rate:** 30.6%

[6] Automated Safety Verification of Programs Invoking Neural Networks
Computer Aided Verification ***(CAV)*** *2021*
Maria Christakis, *Hasan F. Eniser*[†], Holger Hermanns, Joerg Hoffmann, Yugesh Kothari, Jianlin Li, Jorge A. Navas, Valentin Wuestholz
CORE Rank: A* **Acceptance Rate:** ∼21%

[7] Importance-Driven Deep Learning System Testing
International Conference on Software Engineering ***(ICSE)*** *2020*
Simos, Gerasimou, *Hasan F. Eniser*[†], Alper Sen, Alper Cakan
CORE Rank A* **Acceptance Rate:** ∼21%

[8] DeepFault: Fault Localization For Deep Neural Networks
Fundamental Approaches to Software Engineering ***(FASE)*** *2019*
Hasan F. Eniser[†], Simos Gerasimou, Alper Sen.
CORE Rank: B **Acceptance Rate:** 32.4%

[9] Phish-hook: Detecting Phishing Certificates Using Certificate Transparency Logs
International Conference on Security and Privacy in Communication Networks ***(SecureComm)*** *2019*
Edona Fasllija, *Hasan F. Eniser*, Bernd Pruenster
CORE Rank: C **Acceptance Rate:** Unknown

[10] Fancymock: Creating Virtual Services From Transactions
Symposium on Applied Computing ***(SAC)*** *2018*
Hasan F. Eniser[†], Alper Sen, Suleyman Olcay Polat
CORE Rank: Multiconference (Previously B) **Acceptance Rate:** Unknown
Part of MSc Thesis

[11] Temporal Logic Motion Planning Using POMDPs with Parity Objectives: Case Study Paper
Hybrid Systems: Computation and Control ***(HSCC)*** *2015*
Maris, Svorenova, Martin Chmelik, Kevin Leahy, *Hasan F. Eniser*, Krishnendu Chatterjee, Ivana Cerna, Calin Belta
CORE Rank: N/A **Acceptance Rate:** Unknown

Peer-Reviewed Workshop Publications

[1] Synthesizing a Progression of Subtasks for Block-Based Visual Programming Tasks
Workshop on AI for Education **(AI4ED@AAAI)** *2024*
Alperen Tercan, Ahana Ghosh, *Hasan Ferit Eniser*, Maria Christakis, Adish Singla

[2] DeepSmartFuzzer: Reward Guided Test Generation For Deep Learning
Workshop on Artificial Intelligence Safety **AISafety@IJCAI** *2020*
Samet Demir, *Hasan F. Eniser*, Alper Sen

[3] Testing Service Oriented Architectures Using Stateful Service Virtualization via Machine Learning
Workshop on Automation of Software Test **AST@ICSE2019** *2019*
Hasan F. Eniser[†], Alper Sen
Part of MSc Thesis

Technical Reports

[1] Software Bug Detection: Challenges and Synergies
Dagstuhl Seminar 23131, 2023
Marcel Boehme, Maria Christakis, Rohan Padhye, Kostya Serebryany, Andreas Zeller, *Hasan F. Eniser*

[2] Raid: Randomized Adversarial-Input Detection for Neural Networks
arXiv
Hasan F. Eniser[†], Maria Christakis, Valentin Wuestholz

[3] Towards Translating Real-World Code with LLMs: A Study of Translating to Rust
arXiv
Hasan F. Eniser[†], Hanliang Zhang[†], Cristina David, Meng Weng, Brandon Paulsen, Joey Dodds, Daniel Kroening, Maria Christakis

Open Source Research Tools

- **NOMOS** is a declarative, domain-agnostic specification language for writing hyperproperties (i.e. k-safety properties) for ML models. It also comes with an automated framework for validating properties using metamorphic testing. [IJCAI23 Paper] [AAAI24 Paper] [Code]
  Tags: *Python, ANTLR4, Metamorphic Testing, Large Language Models, Code Translation*
- **π-fuzz** implements a fuzzing framework for testing action policies. π-fuzz generates test states via metamorphic relaxations and reveals sub-optimal (buggy) behaviours in action policies. [Code] [ISSTA22 Paper]
  Tags: *Python, PyTorch, Fuzzing*
- **Neuro-Aware Program Analysis** implements a tightly bounded verification process leveraging abstract interpretation for verifying reachability and crash avoidance in C programs invoking neural networks. [Code] [CAV21 Paper]
  Tags: *C, Python, Tensorflow, PyTorch, Abstract Interpretation*
- **DeepImportance** implements a novel relevance-based coverage metric for deep neural networks. We also implemented 5 other coverage metrics from the literature for comparison. [Code] [ICSE20 Paper]
  Tags: *Python, Tensorflow, Coverage criteria*
- **DeepSmartFuzzer** implements a coverage guided fuzzer for neural networks driven by Monte-Carlo Tree Search. [Code] [AISafety@IJCAI20 Paper]
  Tags: *Python, Tensorflow, Coverage Guided Fuzzing, MCTS*
- **DeepFault** implements suspiciousness measures inspired by fault localization in software testing to identify suspicious neurons that are considered responsible for inadequate DNN performance. [Website] [Code] [FASE19 Paper]
  Tags: *Python, Tensorflow, Fault localization*