Automatically Testing Functional Properties of Code Translation Models

Hasan Ferit Eniser[1], **Valentin Wüstholz**[2], **Maria Christakis**[3]

[1] MPI-SWS ,Germany
[2] ConsenSys, Austria
[3] TU Wien, Austria
hfeniser@mpi-sws.org, valentin.wustholz@consensys.net, maria.christakis@tuwien.ac.at

Abstract

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.

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.

Existing work used manually crafted test suites (Rozière et al. 2020) to assess the quality of translations for individual functions. These test suites were later generated automatically using search-based testing (Rozière et al. 2022). However, given the small corpus of functions, it remains unclear how well the models generalize to other code.

Our approach. In this paper, 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 iden-

tifying 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 detected 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, Cheung, and Yiu 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. 2023). 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 paper, 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 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 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.

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

2 Related Work

Code translation. The most closely related work uses LLMs for *code translation* (Rozière et al. 2020; Lachaux et al. 2021; Rozière et al. 2022; Szafraniec et al. 2023) and manually written or automatically generated tests (Rozière et al. 2022) to evaluate semantic correctness of the translated code (similar to our semantic 1-safety property) for a relatively small, curated corpus of programs. In contrast, we focus on testing the correctness of the models themselves. Specifically, we enable NOMOS to express a much broader and more comprehensive set of correctness properties; we also automatically generate new programs, instead of only relying on an existing, curated corpus.

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.

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, Schucher, and Bahdanau 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, Fischer, and Vechev 2023).

3 Approach

In this section, we first give an overview of our specifications for code translation models through examples (Section 3.1). We then describe our testing (Section 3.2) and search (Section 3.3) procedures in detail.

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

(a) Syntactic 1-safety property.

```
1 input pj;
2 requires compiles(pj, "java");
3 output pc;
4 {
5   pc = transpile(pj, "java", "cpp")
6 }
7 ensures compiles(pc, "cpp")
8   ==> retValues(pj, "java") == retValues(pc, "cpp");
```

(b) Semantic 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");
```

(c) Syntactic 2-safety property.

```
1 input pj1;
2 var pj2 := renameParam(pj1, "java");
3 requires pj2 != null;
4 output pp1;
5 output pp2;
6 {
7   pp1 = transpile(pj1, "java", "py")
8   pp2 = transpile(pj2, "java", "py")
9 }
10 ensures compiles(pp1, "py") && compiles(pp2, "py")
11   ==> retValues(pp1, "py") == retValues(pp2, "py");
```

(d) Semantic 2-safety property.

Figure 1: Example k-safety specifications for code translation models.

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.

For a formal description of the language prior to our extension, we refer the reader to the NOMOS paper (Christakis et al. 2023). We describe our extensions in the next section.

First, consider the syntactic 1-safety property shown in Figure 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. Line 1 declares the input program `pj` and line 2 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 lines 6–7, the **`ensures`** clause expresses the postcondition that the number of 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 1b shows a semantic 1-safety property expressing, with a precondition on line 2, 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 1c. On line 1, the property declares an input program `pj1`. Unlike the previous properties, 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. 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 1d. Here, `pj2` is generated from `pj1` by renaming a random function parameter (line 2). 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 (Eniser, Wüstholz, and Christakis 2023).

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, Cheung, and Yiu 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 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 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 (Eniser, Wüstholz, and Christakis 2023).

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.

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.

Algorithm 1: Our search procedure

Input: properties, program P, searchBudget, testBudget, initModelParams
Output: bestTranslation

```
1:  params = initModelParams
2:  // Minimum number of violated properties
3:  minVP = +∞
4:  // Minimum number of total violations
5:  minTV = +∞
6:  bestTranslation = null
7:  while searchBudget > 0
8:    // Current number of violated properties
9:    VP = 0
10:   // Current number of total violations
11:   TV = 0
12:   /* We run the testing procedure for each property
13:   with the current model parameters */
14:   for all prop in properties
15:     /* Function Test returns the number of violations
16:     and the program translation */
17:     v, tr = Test(prop, P, testBudget, params)
18:     TV += v
19:     if 0 < v
20:       VP++
21:   if VP == 0
22:     return tr
23:   if (VP < minVP) ∨ (VP == minVP ∧ TV < minTV))
24:     minVP = VP
25:     minTV = TV
26:     bestTranslation = tr
27:   params = Mutate(params)
28:   searchBudget--
29: return bestTranslation
```

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 1 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 lines 7–28 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 (lines 14–20). If no properties are vio-

lated, the current translation is returned (lines 21–22). Otherwise, on line 23, 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 27 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 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 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");
```

4 Evaluation

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. 2023). 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 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.

4.2 Results for Testing Procedure

We evaluate our testing procedure by checking a total of 38 properties found in (Eniser, Wüstholz, and Christakis 2023).

We introduced many inspection and transformation functions; we now provide a summary of the remaining ones (see (Eniser, Wüstholz, and Christakis 2023) 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 1 and 2 show the percentages of (unique) property violations for all models. The absolute numbers are included in (Eniser, Wüstholz, and Christakis 2023) but are summarized in Table 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

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

Table 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 (Eniser, Wüstholz, and Christakis 2023). 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	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

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

Table 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.

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 (Eniser, Wüstholz, and Christakis 2023).

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.

4.3 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 (Eniser, Wüstholz, and Christakis 2023)) for each translation.

We set the search budget to 20 and the testing budget to 50 (see Algorithm 1). 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 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.

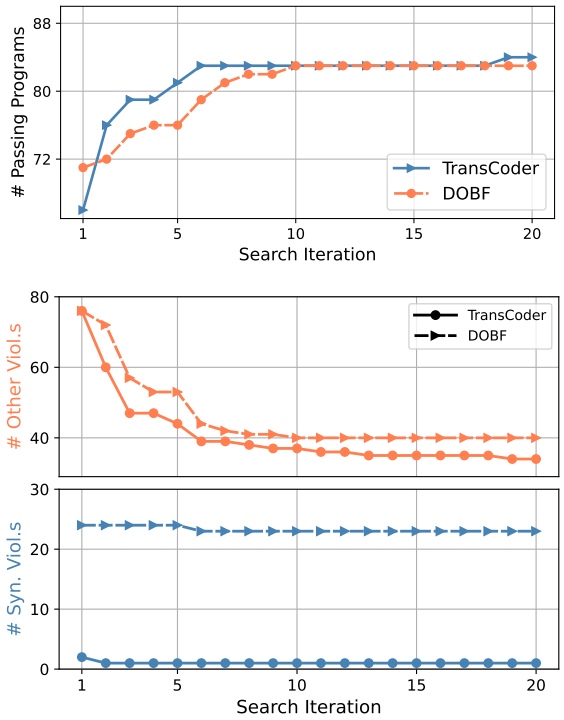

Figure 2: The number of passing programs (top) and the number of syntactic and other property violations (bottom) as the number of search iterations increases from 1 to 20.

RQ2. How long does the search take? As shown in Figure 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 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.

5 Conclusion

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

Acknowledgements

This work was supported by DFG grant 389792660 as part of TRR 248 (see https://perspicuous-computing.science).

References

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.

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.

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.

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*. To appear.

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

Eniser, H. F.; Wüstholz, V.; and Christakis, M. 2023. Automatically Testing Functional Properties of Code Translation Models. *CoRR*, abs/2309.12813.

Lachaux, M.; Rozière, B.; Szafraniec, M.; and Lample, G. 2021. DOBF: A Deobfuscation Pre-Training Objective for Programming Languages. In *NeurIPS*, 14967–14979.

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. 2023. StarCoder: May the Source Be with You! *CoRR*, abs/2305.06161.

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.

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*.

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.

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.

Scholak, T.; Schucher, N.; and Bahdanau, D. 2021. PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models. In *EMNLP*, 9895–9901. ACM.

Segura, S.; Fraser, G.; Sánchez, A. B.; and Cortés, A. R. 2016. A Survey on Metamorphic Testing. *TSE*, 42: 805–824.

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*, 7699–7715. ACM.

Szafraniec, M.; Rozière, B.; Leather, H.; Labatut, P.; Charton, F.; and Synnaeve, G. 2023. Code Translation with Compiler Representations. In *ICLR*. OpenReview.net.

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*, 13818–13843. ACL.