TSDolly: A program generator for TypeScript

Gabriela Araujo Britto
garaujobritto@gmail.com
Federal University of Pernambuco
Recife, PE, Brazil

Leopoldo Teixeira
lmt@cin.ufpe.br
Federal University of Pernambuco
Recife, PE, Brazil

Rohit Gheyi
rohit@dsc.ufcg.edu.br
Federal University of Campina Grande
Campina Grande, PB, Brazil

ABSTRACT

TypeScript is an increasingly popular open-source language that builds on JavaScript by adding optional static type definitions. Its ecosystem has many tools that require confidence in their correctness when manipulating TypeScript programs. Tool developers commonly use tests for increasing confidence in a tool's implementation correctness. To test tool implementations, tool developers can manually write TypeScript programs to be used as test inputs, but they can miss problematic programs as there are many language features to consider. In addition, a range of those tools have properties that apply specifically to programs that compile successfully and satisfy complex constraints. We therefore present a program generation technique that allows generating successfully compiling TypeScript programs, that optionally also satisfy tool-specific constraints not captured by syntactic constraints alone. We evaluated our technique by generating programs in the context of five automated TypeScript refactoring implementation, and by running tests with the generated programs as inputs. The majority (97.45%) of the generated programs compiled without errors, and all of the generated programs could be refactored, meaning that they can indeed satisfy tool-specific constraints to be used as test inputs. We tested the refactorings using the generated programs as test inputs and we found and reported a bug in a TypeScript refactoring, where the refactoring introduced a compilation error to programs that previously had no errors.

CCS CONCEPTS

• **Software and its engineering → Source code generation**.

KEYWORDS

refactoring, program generation, typescript

ACM Reference Format:
Gabriela Araujo Britto, Leopoldo Teixeira, and Rohit Gheyi. 2021. TSDolly: A program generator for TypeScript. In *25th Brazilian Symposium on Programming Languages (SBLP'21), September 27-October 1, 2021, Joinville, Brazil.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3475061.3475079

1 INTRODUCTION

TypeScript is an open-source language which builds on JavaScript by adding static type definitions [17]. We can view it as an optionally typed [4] superset of JavaScript. TypeScript recognizes JavaScript code, extending it with optional type annotations and static type checking, while maintaining JavaScript's semantics. TypeScript is becoming increasingly popular. In 2020, it was the fourth most popular language on GitHub [13], and 78% of the respondents to the State of JavaScript 2020 Survey had used it [15]. Moreover, the TypeScript compiler is capable of detecting a considerable amount of JavaScript bugs during compilation [6].

The TypeScript ecosystem has industrial and academic tools that act on TypeScript programs, such as *Stryker*, a mutation testing tool [16], `rsc` [18], a refinement type-checker for TypeScript, and *WebStorm* [19], an IDE that provides TypeScript refactorings. The TypeScript compiler itself acts on TypeScript programs, type checking and emitting JavaScript code. The TypeScript project aims to provide consistent tool support [17], so the TypeScript compiler also handles most of code editor integration, including code completion and its own automated refactoring implementations.

To evaluate the functionalities and properties of such tools, it is important to have programs without compilation errors. *Stryker*, for instance, considers that if a mutant (i.e. a modification of a program) does not successfully compile, it is ignored [16]. Therefore, mutations should only be applied to programs without compilation errors, otherwise the tool might generate mutants that also have compilation errors and will be ignored. Similarly, it makes more sense for the `rsc` refinement type-checker to be tested with TypeScript programs that successfully compile, because we would like it to detect errors that the TypeScript compiler does not already detect. If we consider automated refactoring implementations for TypeScript, checking if a refactoring preserves program behavior requires us to be able to define a program's behavior, which means this property is also better suited for TypeScript programs that successfully compile.

A commonly used approach to increase confidence on the intended behavior of tool implementations is testing, even though tests cannot formally prove their correctness. To run tests, we typically need test inputs, which in this case, are compilable TypeScript programs. A possible approach for tool developers is to manually write programs to be used as test inputs. However, this can be a difficult task, since developers need to consider many language features and the interaction of those features to decide what programs to use for testing, relying on their intuition and previous experience.

To address this problem, techniques for automated program generation have been developed. Program generators have been successfully used in the context of C and Java to test compilers and refactoring implementations [3, 14, 20]. Some of the proposed program generation approaches are based on language grammars [3, 20], and do not aim to generate programs that successfully compile or that satisfy other arbitrary complex constraints. However, as mentioned, there are tools whose tests require compilable programs and programs that satisfy certain properties not easily described based on syntactic rules alone. Other approaches do aim to generate successfully compiling programs [8, 10, 14], but they do not focus on generating TypeScript programs [10, 14], or cannot generate programs that satisfy arbitrary, developer-defined constraints [8], so they are not suited for testing tools targeting TypeScript.

In this work, we present a program generation technique that allows generating successfully compiling TypeScript programs, that optionally also satisfy complex, user-defined constraints (Section 3), not captured by syntactic rules. Our technique adapts JDolly [14], a tool that generates compilable Java programs that also satisfy additional constraints required for testing. We implement a tool called TSDolly, which specifies a subset of the TypeScript language (Section 3.1) as a foundation for generating TypeScript programs. We write this specification using Alloy [2], a declarative modeling language based on the concepts of sets and relations. Moreover, we can add additional constraints in the specification, to further guide program generation. For instance, we can ensure that all generated programs must have at least one class with more than one field.

TSDolly uses the Alloy Analyzer, a model checker and finder tool, to generate instances (i.e. bindings of variables to values) that satisfy a specification (Section 3.2). Our specification describes TypeScript programs, so those instances are Alloy's encoding of TypeScript programs. TSDolly then transforms those instances into TypeScript programs (Section 3.3), which can be fed as test inputs to other tools.

We evaluated TSDolly under two dimensions. First, we evaluate whether it generates programs that successfully compile. We do so by considering five refactoring implementations provided by the TypeScript compiler (Section 4.2.1), generating programs that satisfy the preconditions of such refactorings (Section 4.2.2). For each refactoring, we generated between 250,000 and 546,000 different Alloy instances that encode programs (Section 4.3), and a sample of those instances were transformed into TypeScript programs. More than 97.45% of the generated programs compiled without error.

We also evaluated whether these programs could be usefully fed as inputs to test the refactoring implementations (Section 4.2.3). We applied the refactorings to the generated programs, and found that all of them could be refactored (Section 4.3). We also found and reported a bug in one of those refactoring implementations, which was later fixed by the TypeScript compiler team.

2 MOTIVATING EXAMPLE

Refactoring is the process of modifying code with the goal of improving its internal structure without altering its external behavior [5], improving code readability and organization and making it easier to maintain. To aid the programmer in the process of applying common refactorings, many IDEs provide refactoring implementations that automate this process. However, refactoring developers must be careful that their refactoring implementation is correct with respect to behavior preservation: a refactoring should not change the program's behavior, otherwise it could introduce bugs into the program.

A common approach to increase confidence in their refactoring implementation correctness is to use testing [12, 14]. To test refactorings, refactoring developers need to use programs as test inputs. Additionally, to test if a refactoring preserves program behavior, developers need *compilable* programs, because the notion of program behavior is typically not well-defined for programs with compilation errors as they might have undefined behavior.

To test automated TypeScript refactoring implementations, refactoring developers can manually write programs to be used as test input, relying on their intuition and experience when selecting programs. However, they might miss programs that could reveal bugs in refactoring implementations. With that in mind, it is important to have a complementary, systematic approach for generating TypeScript programs for testing refactorings.

There are 152 reported and confirmed unique bugs in the refactorings provided by the TypeScript compiler, 40 of which have not yet been fixed,[1] and there might be additional, unreported bugs. As an example, consider the "Extract Symbol" refactoring provided by the TypeScript compiler, which encompasses the "Extract function", "Extract method", and "Extract variable" refactorings [5]. This refactoring can extract an expression (a piece of code) into a new variable, function or method, and this new variable, function, or method can then be used in the place of the extracted expression. If we apply the refactoring to line 2 (highlighted in blue) of Listing 2.1, we extract that expression into the global `new_function`. In the resulting program, shown in Listing 2.2, when `new_function` is called in line 2, `normal_arg` is passed as an argument, preserving the original behavior of the program. However, applying the "Extract Symbol" refactoring to line 3 (highlighted in orange) of Listing 2.1 results in Listing 2.3, which does not have the same behavior: the new program prints **`undefined`** and `"that"`, whereas the original program prints `"this"` and `"that"`.

```
1 function test_extract(this: any, normal_arg: string) {
2     console.log(normal_arg);
3     console.log(this);
4 }
5 test_extract.call("this", "that");
```

Listing 2.1: TypeScript program example.

```
1 function test_extract(this: any, normal_arg: string) {
2     new_function(normal_arg);
3     console.log(this);
4 }
5 function new_function(normal_arg: string) {
6     console.log(normal_arg);
7 }
8 test_extract.call("this", "that");
```

Listing 2.2: Result from refactoring line 2 (highlighted in blue) of program in Listing 2.1.

This happens because **`this`** is a special, implicit function argument. When a function or method is called, **`this`** is implicitly bound to an object at run time according to a set of rules. TypeScript, however, allows a first parameter called **`this`** in function declarations, solely to declare its expected type, and the parameter is erased when compiling to JavaScript. We can explicitly bind **`this`** by using the `call` method (all JavaScript/TypeScript functions are objects), as in

[1]According to the TypeScript GitHub repository's list of issues, as queried in May 20, 2021.

the last line of the example programs. `test_extract.call("this", "that")` means calling `test_extract` with **this** bound to the string `"this"`, and `"that"` as the regular `normal_arg` argument.

In the second refactored program (Listing 2.3), when we call `new_function` in line 3, the **this** in `new_function` does not refer to the same object as the **this** in `test_extract`. In the original program, `test_extract.call("this", "that")` means we are calling `test_extract` with **this** bound to `"this"`, and `normal_arg` as `"that"`, but in Listing 2.3, when we call `new_function()` in `test_extract`'s body, we do not explicitly set `new_function`'s **this**. As per JavaScript rules, `new_function`'s **this** is undefined,[2] instead of the string `"this"`, so behavior is not preserved.

```
function test_extract(this: any, normal_arg: string) {
    console.log(normal_arg);
    new_function();
}
function new_function() {
    console.log(this);
}
test_extract.call("this", "that");
```

Listing 2.3: Result from refactoring line 3 (highlighted in orange) of program in Listing 2.1.

When this bug was reported, the existing tests did not catch this case, caused by the special nature of **this**. This illustrates the need for a systematic, automated technique for generating complex TypeScript programs suited to be used as test inputs for TypeScript tools, which can address corner cases that might be difficult to tackle with manually written test inputs. We therefore propose a program generation technique that allows generating TypeScript programs that satisfy complex constraints.

3 TSDOLLY

TSDolly[3] is a tool that implements our technique for automated TypeScript program generation. It uses Alloy [9], a declarative specification language, to model a subset of the TypeScript language. We also use the Alloy Analyzer tool [9] to generate all programs up to a certain size that satisfy this TypeScript specification and additional constraints that might be useful for generating programs with particular characteristics. The TypeScript programs we generate can then be used as test inputs for testing TypeScript language tools, such as automated refactoring implementations, compilers, and other tools that require complex program inputs.

Figure 1 shows the main steps of TSDolly. It receives as input a TypeScript language specification file, and optionally, predicates to further constraint program generation, and a number called skip, which determines how many Alloy instances we want to sample for generating TypeScript programs. In step 1 (instance generation), TSDolly uses the Alloy Analyzer API to generate Alloy instances from a TypeScript specification and additional predicates. In step 2 (building TypeScript programs), TSDolly transforms those instances into TypeScript programs using the TypeScript compiler API. The skip value is used in this step to select a sample of instances to ensure TSDolly runs under time constraints.

[2]In JavaScript's strict mode. In non-strict mode, **this** defaults to the global object.

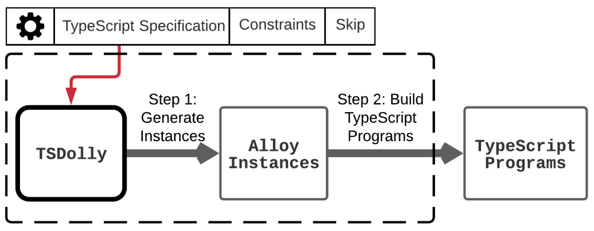

Figure 1: TSDolly architecture.

3.1 TypeScript specification in Alloy

We use Alloy, a language and analyzer used for software modeling [1, 2, 9]. The Alloy language is declarative and based on the concept of atoms, which are primitive entities of a specification, and relations between sets of atoms, where a relation is a set of atom tuples. Programs generated by TSDolly are modeled by a specification of a subset of the TypeScript language. In this specification, we model the tree structure that corresponds to the TypeScript AST. To specify an AST node, we use Alloy's signatures: a signature defines a set of atoms and relations involving those atoms, so we declare one signature for each AST node type included in our specification. We prioritized modeling language features that are relevant for the tests we performed to evaluate TSDolly, as described in Section 4.

For instance, the `FunctionDecl` signature in Listing 3.1 specifies a function declaration node. It extends an abstract signature `Declaration`, which encompasses all top-level declarations, that is, functions or classes. A function declaration has exactly **one** identifier, **one** code block, and the `parameters` relation establishes that it has zero or more parameters. We model parameter lists with Alloy sets [9], resulting in a simple and efficient specification, without harming the compilation rate, as discussed in Section 4. This is similar to the TypeScript AST, where a function declaration node has a list of parameter nodes as children.

```
sig FunctionDecl extends Declaration {
  name: one FunctionIdentifier,
  parameters: set ParameterDecl,
  body: one Block
}
```

Listing 3.1: Alloy specification of function declarations.

We focused on generating programs that compile under the strictest type checking rules enforced by the TypeScript compiler. We avoid syntax errors by modeling TypeScript ASTs, and to avoid most semantic errors, we specify constraints that approximate semantic rules enforced by the TypeScript compiler. Those are declared using Alloy's facts, which establish properties that are always true and must always be satisfied. For instance, Listing 3.2 illustrates the fact that any function being called in the generated program should have been declared.

We can also use predicates to further guide program generation by specifying additional constraints that generated programs must

[3]TSDolly's implementation can be found in https://github.com/gabritto/tsdolly/

```
fact FunctionCalledExists {
  all c: FunctionCall | some f: FunctionDecl | c.name = f.name
}
```

Listing 3.2: Alloy fact stating that functions called in the program must exist.

satisfy. Unlike a fact, an Alloy predicate is a property of the model that does not have to be true. Instead, a predicate is a property that we can use when convenient. We can instruct Alloy to generate instances that satisfy a certain predicate. For instance, in our evaluation (Section 4), we test automated refactoring implementations. Therefore, we specify the refactoring preconditions as additional constraints to generate programs that are suitable to be refactored.

Consider the **Convert parameters to destructured object** refactoring, provided by the TypeScript compiler and illustrated in Listing 3.3. Its goal is to emulate the ability to use named parameters, since TypeScript does not have builtin support for it. It transforms the parameter list of the function into a single parameter object, where each of the object's property corresponds to an old parameter. The function can then be called with an argument object, where each property of this argument object corresponds to a named argument.

```
function getName(firstName: string, lastName: string) {
  return firstName + " " + lastName;
}
getName("Barbara", "Liskov");
```

```
function getName({firstName: string, lastName: string}) {
  return firstName + " " + lastName;
}
getName({firstName: "Barbara", lastName: "Liskov"});
```

Listing 3.3: "Convert parameters to destructured object" refactoring example. At the top is the original program. At the bottom is the result of applying the refactoring.

Listing 3.4 shows the `ConvertParamsToDestructuredObject` predicate, which establishes the refactoring precondition. It states that our program instance should have either a function or a method declaration atom with more than one parameter. Such predicates can guide program generation towards a specific goal.

3.2 Instance generation

We use the Alloy Analyzer tool to generate instances that satisfy our TypeScript specification and additional constraints. Alloy Analyzer is a model finder: it finds a binding of the variables to values (i.e. an instance) that satisfies the specification [2], if it exists. Alloy Analyzer exhaustively searches for instances in a finite search space. It

```
pred ConvertParamsToDestructuredObject {
  (some f: FunctionDecl | #f.parameters
  ↪  > 1) or (some m: MethodDecl | #m.parameters > 1)
}
```

Listing 3.4: Predicate with the necessary conditions of the "Convert parameters to destructured object" refactoring.

finds all instances satisfying a specification, but only up to a certain size, denoted by signature scopes. A scope is a number that indicates the upper bound size of the instances Alloy Analyzer can generate. The scope describes how many atoms of a given signature there can be in an instance. A scope of two means, in our case, that an instance can only have at most two declarations: two functions, two classes, one function and one class, and so on.

To generate instances for our TypeScript specification, we run the Alloy Analyzer through the Alloy API. These instances represent TypeScript programs that TSDolly later converts into actual TypeScript programs. Figure 2 shows an instance generated by Alloy in this step. The instance generation step starts by taking as arguments the TypeScript specification and additional constraints. Then, TSDolly calls the Alloy API to find all instances, up to a certain size, that satisfy our specification and constraints. As a result, the Alloy API returns an iterator that contains the desired instances.

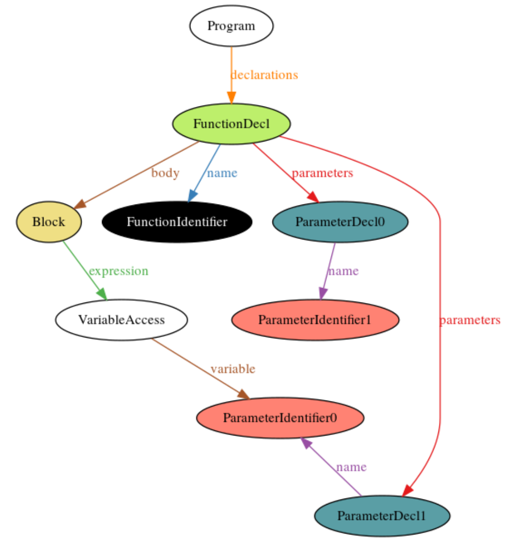

Figure 2: An Alloy instance representing a TypeScript program. The program has a function declaration with two parameters, and its body has an access expression of the first parameter. Type annotations are omitted for conciseness.

We use Java to implement the instance generation step, since the Alloy API is also implemented in Java. In the following step we use TypeScript, because, similarly, we want to use the TypeScript compiler API. This means we must communicate the instances generated in this step, implemented in Java, to the next step, implemented in TypeScript. To that end, as we iterate through the Alloy instances, we serialize them into JSON strings, written into a file, which is used by the next TSDolly step.

3.3 Building TypeScript programs

Given an Alloy instance, we convert it into an actual TypeScript program that can be used as test input by TypeScript tools. We start this step by reading the serialized instances produced by the previous step. We use the `JSON.parse` method from JavaScript's standard library to parse the strings into TypeScript objects. Each object is a representation of an Alloy instance generated in step 1.

We then build a TypeScript AST from each object, following the AST structures used by the TypeScript compiler. An Alloy instance object has a tree shape, since the TypeScript specification written in Alloy describes TypeScript ASTs. More precisely, an Alloy instance object corresponds to the root node of a TypeScript AST, and its properties corresponds to pointers to its children nodes. The same applies for each object pointed to by the Alloy instance object's properties, and, recursively, to each object in the tree rooted at the Alloy instance object: each one corresponds to an AST node. We convert an instance object into an AST by doing a tree traversal and building the corresponding AST node for each visited object, using the TypeScript compiler API. The complexity of this task is reduced by the fact that our specification follows the AST structure used by the TypeScript compiler. At the end of this process, we have the AST for a TypeScript program. Listing 3.5 shows the TypeScript program generated from the Alloy instance in Figure 2.

```
function fun(param0: number, param1: number) {
   param0;
}
```

Listing 3.5: Result of converting the generated Alloy instance in Figure 2 into a TypeScript program.

Due to the nature of programming language specifications and the bounded-exhaustive instance generation, step 1 may generate too many instances. If we try to further limit the bounds, we may still get too many programs, or generating trivial programs that might not catch bugs arising from the interaction among multiple AST nodes. Thus, we apply a skip-based strategy for sampling Alloy instances. Among its arguments, TSDolly receives a positive integer *skip*, which indicates how we should sample instances. For a skip value of *n*, the sampling strategy consists in taking contiguous sequences of size *n* of Alloy instances, and randomly selecting one of them. The sampled instances are converted into TypeScript programs as previously described, and only those programs are used by the next step. The remaining instances are ignored.

Using this skip-based strategy reduces execution time (Section 4.3). It also avoids the generation of many similar programs, since the Alloy Analyzer typically generates many similar, contiguous instances. Previous uses of this technique [12] indicate that the risk of missing bug-revealing instances is small and far outweighed by the performance gains.

4 EVALUATION

TSDolly aims to generate TypeScript programs that compile successfully and can be used as test input for TypeScript tools. Therefore, our evaluation is structured into two parts. We first evaluate the feasibility of using TSDolly, in terms of generating compilable programs. We generate programs according to five refactoring implementations provided by the TypeScript compiler as our target application. The second part of the evaluation addresses usefulness, using the generated programs as inputs to test those refactoring implementations. In what follows, we discuss our research questions, the study planning, and our results and discussion.

4.1 Research Questions

We first evaluate the program generation aspect of TSDolly. We do so with respect to its ability to generate successfully compiling programs, as well as its performance, in the context of refactoring implementations from the TypeScript compiler. We address the following research questions:

RQ1. *What is the compilation rate of TSDolly-generated programs?* We measure the number of Alloy instances generated in step 1; the number of instances that were sampled to be transformed into TypeScript programs in step 2 and used as test inputs in RQ3; and the number of TypeScript programs produced by step 2 that successfully compile. Those metrics reflect our goal of generating programs that can be used as test inputs for TypeScript tools.

RQ2. *What is the average time needed for generating TypeScript programs with TSDolly?* We measured the execution time for all refactoring implementations used for generating programs.

In the second part of our evaluation, we evaluate whether TSDolly generates useful programs to be fed as inputs to evaluate TypeScript tools. We do so by testing refactoring implementations with respect to evaluating if compilation behavior is preserved. We address the following research question:

RQ3. *Can TSDolly generate programs that are useful as test inputs for TypeScript refactoring implementations?* We measure the number of TypeScript programs that could be refactored at least once by each target refactoring, i.e. the number of refactorable programs. We registered the generated programs, the compilation errors, and test failures. The number of refactorable programs tells us how many of the generated programs can be used as test inputs.

4.2 Study Planning

4.2.1 Subject selection. We generate TypeScript programs specifically targeted at testing refactoring implementations. We use five refactoring implementations, out of the nine implemented by the TypeScript compiler version 3.9.5. Besides "Convert parameters to destructured object" (Section 3.1), we also consider "Generate 'get' and 'set' accessors", also known as "Encapsulate Field" [5, Chapter 6], which generates getter and setter methods (accessors) for a class field; "Convert to template string", which converts a string concatenation expression to a template literal; "Extract Symbol", which extracts certain expressions into a new symbol (similar to "Extract Variable" and "Extract Function" [5, Chapter 6]); and "Move to a new file", which moves a top-level declaration to a new file.

We can add constraints to our specification to guide program generation (Section 3.1). A test input for TypeScript refactoring implementations must be able to be refactored by the refactoring under test. In other words, the generated programs must satisfy the refactoring preconditions. To this end, we modeled the preconditions of each of the five tested refactorings as Alloy predicates, and added those predicates to our specification to generate programs suitable to be refactored (see **RQ3**).

4.2.2 Setup for program generation. To address **RQ1** and **RQ2**, we generate TypeScript programs for testing refactorings. We ran TSDolly with a set of parameters, some of which were fixed throughout all runs, while some changed.

Fixed parameters. The TypeScript specification we used was fixed, containing both the syntactic and semantic rules of TypeScript programs, and the refactoring preconditions. The scope, that is, the upper bound size, of the generated Alloy instances, was also fixed and always equal to two. We chose a scope of two because a scope of one only generates trivial programs. In an instance generated with scope one, there cannot be two classes where one extends the other, but a scope larger than two generated millions of instances, far more than we had the resources to process for testing.

Settings for the TypeScript compiler also remained the same. We used the strictest level of type checking rules, and disabled code emitting, using the compiler only for type checking and refactoring. We ran the experiments on a computer running Microsoft Windows 10 Home, 16GB RAM, Intel Core i7-7700HQ @ 2.80Hz.

Variable parameters. We targeted a single refactoring at a time, so we separately ran TSDolly for each of the five refactorings tested. For each refactoring, we ran TSDolly twice, once with a skip value of 25 and once with a skip value of 50. Choosing a skip value of 25 and 50 means that 4% ($\frac{1}{25}$) and 2% ($\frac{1}{50}$) of the Alloy instances generated in step 1 were transformed into a TypeScript program.

Although we would have liked to process all of the generated Alloy instances, we chose those values because we had limited resources to run TSDolly. So we needed to sample the generated instances, and we chose the specific values of 25 and 50 because the execution time of TSDolly could then fit our time constraint. Moreover, the value of 25 has been used in a previous study that used a program generator (JDolly) also based on Alloy [11].

Since we ran TSDolly twice for each refactoring, changing only the skip value, we were able to reuse the Alloy instances generated in step 1, because this step depends only on the target refactoring and not on the value of the skip argument. For each refactoring, we first ran the instance generation (step 1) of TSDolly. The generated instances were serialized and saved, and then read and used as input when we ran step 2 (building TypeScript programs) of TSDolly's pipeline with skips of 25 and 50.

4.2.3 Testing TypeScript refactorings. To address **RQ3**, we investigate how useful TSDolly-generated programs are for testing TypeScript refactoring implementations. Given a TypeScript program generated for a specific refactoring, we first check whether this program is refactorable. To test a refactoring, we apply it to the input program, and then compare both program versions to check if behavior is preserved. In this evaluation, we only check if compilation behavior is preserved, that is, if programs compiling without errors still continue to do so after being refactored. We made this choice because we had the TypeScript readily accessible to be used as a test oracle for checking this property.

We use the TypeScript language service API, provided by the TypeScript compiler, to apply refactorings to the generated programs. For each generated program, we first find out if the refactoring is applicable to the program, and, if so, to what program position(s). The `getApplicableRefactors` function receives a program and a position in that program and returns a list of refactorings that can be applied to the specified position. To avoid calling this complex function for every program position, we use node predicates (i.e. functions that take a TypeScript AST node and return true or false) to determine if we should actually call it. For each refactoring under test, we have a predicate that returns true for all AST nodes where this refactoring is applicable. For instance, "Generate 'get' and 'set' accessors" can only be applied to class field declarations, so this refactoring's predicate only returns true for such nodes. We traverse the AST, calling the predicate on each node. If the predicate returns true, we call `getApplicableRefactors` with the program and that node's position in the program.

We then get a list of applicable refactorings and their respective program positions. We ignore other refactorings besides the one under test. We then call `getEditsForRefactor` for each program position where the target refactoring is applicable. This function receives a program, a position in that program, and a desired refactoring, and returns the changes resulting from applying the refactoring to that program position. After calling it for every applicable program position, we have a list of changes. Since applying a refactoring to different program positions might result in the same changes, we also remove duplicates from this list.

To test the refactoring, we first compile the input program and collect possible compilation errors. Then, we apply each possible refactoring change to the program, obtaining a set of new, refactored programs. We also compile the refactored programs and collect their compilation errors. We compare the compilation errors of the original program and its refactored counterparts. If the original program has no compilation errors, but a refactored program has errors, then we consider the test failed; otherwise, the test passed.

4.3 Summary of the Results

Table 1 presents the results of the program generation metrics collected in our experiments. Regarding **RQ1**, for each refactoring, TSDolly generated between 250,000 and 546,000 different Alloy instances. We sampled 4% and 2% (skip 25 and 50) of those instances to be transformed into TypeScript programs and compiled. The percentage of successfully compiling programs was high, above 97.45% for all refactorings, regardless of the skip value.

With respect to **RQ2**, Figure 3 shows time measurements from running TSDolly, for a skip of 25. Results for a skip of 50 had similar average times and proportional total times and thus are omitted for conciseness. Generating Alloy instances (step 1 of TSDolly) took, on average, less than two milliseconds for each generated instance, and a total time between three and eleven minutes across all refactorings. Building TypeScript programs took, on average, close to one millisecond and a total time under one second. Meanwhile, compiling the programs took, on average, between one and two seconds for each program, and compiling all the programs took up to eleven hours, dominating the execution time of TSDolly.

We also measured the time it took to run the tests for each refactoring (RQ3), using the generated programs as input. This includes the time to apply the refactoring to the generated programs, compiling the refactored programs, and checking if the refactoring introduced any compilation error to previously compilable programs. The average time per test ranged between two and eleven seconds, whereas total time, for a skip of 25, ranged between 6 and 52 hours.

Finally, considering **RQ3**, Table 1 shows that all of the generated programs could be refactored. We expected this as we included the refactoring preconditions in our specification.Our specification describes a subset of TypeScript, so the generated programs did not trigger preconditions that would prevent a refactoring from being applied. When adding a new language feature to our specification,

Table 1: Program generation metrics from TSDolly experiments. 'Generated instances' is the number of Alloy instances generated in step 1. 'Sampled' is the number of Alloy instances sampled for steps 2 and 3. 'Compilable' is the number of TypeScript programs produced from the sampled instances that successfully compile. 'Refactorable' is the number of programs produced from the sampled instances that are refactorable at least once by the target refactoring.

Refactoring	Generated instances	Sampled		Compilable		Refactorable	
		Skip 25	Skip 50	Skip 25	Skip 50	Skip 25	Skip 50
Convert parameters to destructured object	315,310	12,612 (4%)	6,306 (2%)	12,379 (98.15%)	6,183 (98.05%)	12,612 (100%)	6,306 (100%)
Convert to template string	239,399	9,576 (4%)	4,788 (2%)	9,491 (99.11%)	4,748 (99.16%)	9,576 (100%)	4,788 (100%)
Generate 'get' and 'set' accessors	546,968	21,879 (4%)	10,939 (2%)	21,342 (97.55%)	10,660 (97.45%)	21,879 (100%)	10,939 (100%)
Extract Symbol	406,096	16,244 (4%)	8,122 (2%)	15,854 (97.60%)	7,935 (97.70%)	16,244 (100%)	8,122 (100%)
Move to a new file	499,038	19,962 (4%)	9,981 (2%)	19,522 (97.80%)	9,755 (97.74%)	19,962 (100%)	9,981 (100%)

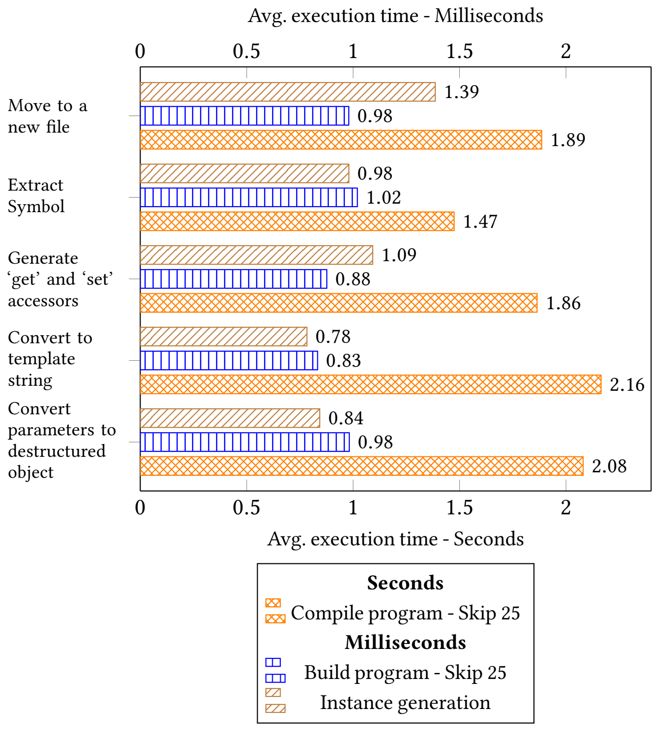

Figure 3: Average time for generating an instance (step 1), building a program (step 2), and compiling a program.

we can adjust the refactoring preconditions to account for the new feature so that most of the generated programs can be refactored.

4.4 Discussion

In this section, we discuss the results of our evaluation.

Compilation errors. The compilation rate is inferior to the refactorable programs rate, because the refactoring implementations allow refactoring programs with compilation errors, if they satisfy the preconditions. Our specification does not include all type checking rules. We focused on adding only enough rules to reach a high compilation rate, prioritizing the simplicity of the specification. Nevertheless, we inspected the compilation errors to understand why some programs do not compile. We identified two kinds of errors.

The first kind occurs when class `B` extends class `A`, overrides one of `A`'s methods, but the type of the overriding method is not assignable to the type of the overridden method. The compiler considers the parameter order when checking type assignability. We model parameter lists with sets, so we cannot fully express assignability rules.

The other kind occurs when the type of an argument of a method or function call is not assignable to the declared type of the corresponding parameter. Our specification does not include constructors yet, so it cannot initialize object fields. To avoid the compiler emitting non-initialized field errors for non-optional fields, we constrain all fields to be declared as optional. This error arises in generated programs when an object's field is used as an argument, since the field is optional and the argument is not, making their types incompatible.

These two kinds of compilation errors could be eliminated by extending or modifying our TypeScript specification. Still, the vast majority of the generated programs are compilable, so the choice for simplicity did not compromise our program generation goals.

```
class ClassIdentifier_0 {
   private Field_1?: string;
}
```

```
class ClassIdentifier_0 {
   private _Field_1?: string;
   public get Field_1(): string {
      return this._Field_1;
   }
}
```

Listing 4.1: "Generate 'get' and 'set' accessors" refactoring bug. Top shows original program. Bottom shows program after refactoring, where line 4 triggers a compilation error.

Performance. Figure 3 shows that using a sampling strategy is justified. Instance generation and program building take minutes. The value of the skip argument, which determines the sample size, can be adjusted according to resource constraints so that the duration of compilation and tests is compatible with those constraints.

Usefulness of generated programs. There were no uncaught exceptions when calling the TypeScript refactoring API, meaning that checking the availability of refactorings and applying refactorings did not crash for the generated programs.

Bug found. Our tests uncovered a bug in the "Generate 'get' and 'set' accessors" refactoring, which introduced compilation errors to previously compilable programs. The compilation error was introduced when the refactoring was applied to a field marked as optional and generated a getter function with an inappropriate return type.

Listing 4.1 shows an example. `Field_1` is annotated with type `string` and marked as optional (question mark). Because this field is optional, it might not be set to any string, and its value will be `undefined`, so, to the TypeScript compiler, this field's type is a union of the `string` and `undefined` types. Applying the refactoring generates the `Field_1` getter function, whose return type should be the same as the field's type, but instead is `string`. The compiler yields an error with the message "Type 'string | undefined' is not assignable to type 'string'". We reported this behavior to the TypeScript team,[4] they confirmed it was a bug, and it has since then been fixed.

5 RELATED WORK

Our work is heavily inspired by JDolly [14], a tool that generates Java programs using an Alloy specification of Java and the Alloy Analyzer to generate instances that are then transformed into Java programs that were used to test refactoring implementations in popular IDEs [11, 14]. This technique has also been adapted to generate C programs through CDolly [12]. Our work differs from JDolly in that we focus on generating compilable TypeScript programs instead of Java programs, and had to adapt the technique to this language.

Kreutzer et al. [10] propose a language-agnostic technique that takes semantic rules into consideration to generate programs. Users specify syntactic and semantic rules of a language using a language based on attribute grammars. Their goal is broader, since they are interested in evaluating compilers. We take a focused approach as we model a subset of the TypeScript language and so far, we only evaluated TSDolly in the context of testing refactoring implementations.

CodeAlchemist [8] is a fuzz testing tool that generates JavaScript programs that avoid semantics errors. It combines fragments from existing JavaScript programs following specific rules to avoid semantic errors, using the semantics-aware assembly technique. It successfully found bugs in JavaScript engines. CodeAlchemist uses a seed-based fuzzing approach, so the quality of the generated programs depends on seed selection, and it focuses on generating semantically valid JavaScript. In contrast, our approach allows us to encode arbitrary constraints in our specification, in addition to semantic validity. We exhaustively generate programs that satisfy such constraints up to a given bound, not relying on seed programs, and use random sampling to select which generated programs to use for testing.

Gligoric et al. [7] used eight open-source projects to test Java and C refactorings from Eclipse, successfully finding new bugs. Using existing programs avoids the need for a program generator, and any bugs found have the benefit of occurring in real programs.However, using real programs can increase the time, space, and human costs of checking correctness, especially in large programs. A further complication is the diversity of tools and standards in the TypeScript ecosystem. We may need to be mindful of what environment a project targets (e.g. browsers or the Node.js runtime), the language version, etc., but we can control such aspects by using generated programs as test inputs.

6 CONCLUSIONS

In this work, we propose TSDolly, a TypeScript program generator, that generates programs that are compilable and that can be used as test inputs to evaluate TypeScript-related tools. We evaluate TSDolly to generate programs focused on five refactoring implementations from the TypeScript compiler. The majority of the generated programs (97.45%) compile without errors, and all of them can be refactored by the refactorings under test. By combining our program generation technique with the TypeScript compiler as a test oracle for checking compilation behavior preservation, we were able to find a bug in one of the refactoring implementations, which we reported, and it has since then been fixed. This provides initial evidence that TSDolly can be used to automatically generate programs that compile and are useful for evaluating TypeScript tools, complementing the use of other techniques such as manually writing tests. As future work, we would like to include more language features into our specification and evaluate it with different tools and more refactorings.

ACKNOWLEDGMENTS

We acknowledge support by FACEPE (APQ-0570-1.03/14), CNPq (426005/2018-0 and 311442/2019-6), CAPES (117875 and 175956), and FAPEAL (60030.0000000462/2020). This work is partially supported by INES 2.0,FACEPE grants PRONEX APQ-0388-1.03/14 and APQ-0399-1.03/17, and CNPq grant 465614/2014-0.

[4]Bug report: https://github.com/microsoft/TypeScript/issues/40994.

REFERENCES

[1] Alloy API [n.d.]. Alloy API documentation. URL: http://alloytools.org/documentation/alloy-api/index.html. Accessed: 21 May 2021.

[2] Alloy tools [n.d.]. Alloy tools. URL: https://alloytools.org. Accessed: 21 May 2021.

[3] Gergö Barany. 2018. Liveness-Driven Random Program Generation. In *Logic-Based Program Synthesis and Transformation*, Fabio Fioravanti and John P. Gallagher (Eds.). Springer International Publishing, Cham, 112–127.

[4] Gilad Bracha. 2004. Pluggable type systems. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*.

[5] Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[6] Zheng Gao, Christian Bird, and Earl T Barr. 2017. To Type or Not to Type: Quantifying Detectable Bugs in JavaScript. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 758–769. https://doi.org/10.1109/ICSE.2017.75

[7] Milos Gligoric, Farnaz Behrang, Yilong Li, Jeffrey Overbey, Munawar Hafiz, and Darko Marinov. 2013. Systematic Testing of Refactoring Engines on Real Software Projects. In *ECOOP 2013 – Object-Oriented Programming*, Giuseppe Castagna (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 629–653.

[8] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society.

[9] Daniel Jackson. 2012. Software Abstractions - Logic, Language, and Analysis, Revised Edition. *The MIT Press* (2012).

[10] P. Kreutzer, S. Kraus, and M. Philippsen. 2020. Language-Agnostic Generation of Compilable Test Programs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 39–50. https://doi.org/10.1109/ICST46399.2020.00015

[11] M. Mongiovi, R. Gheyi, G. Soares, M. Ribeiro, P. Borba, and L. Teixeira. 2018. Detecting Overly Strong Preconditions in Refactoring Engines. *IEEE Transactions on Software Engineering* 44, 5 (2018), 429–452. https://doi.org/10.1109/TSE.2017.2693982

[12] Melina Mongiovi, Gustavo Mendes, Rohit Gheyi, Gustavo Soares, and Márcio Ribeiro. 2014. Scaling Testing of Refactoring Engines. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. IEEE Computer Society, 371–380. https://doi.org/10.1109/ICSME.2014.59

[13] octoverse2020 [n.d.]. The State of the Octoverse | The State of the Octoverse explores a year of change with new deep dives into developer productivity, security, and how we build communities on GitHub. URL: https://octoverse.github.com/. Accessed: 7 May 2021.

[14] G. Soares, R. Gheyi, and T. Massoni. 2013. Automated Behavioral Testing of Refactoring Engines. *IEEE Transactions on Software Engineering* 39, 2 (2013), 147–162. https://doi.org/10.1109/TSE.2012.19

[15] stateofjs2020 [n.d.]. State of JS 2020: JavaScript Flavors. URL: https://2020.stateofjs.com/en-US/technologies/javascript-flavors/. Accessed: 7 May 2021.

[16] Stryker [n.d.]. Stryker Mutator. URL: https://stryker-mutator.io. Accessed: 21 May 2021.

[17] TypeScript [n.d.]. TypeScript. URL: https://www.typescriptlang.org/. Accessed: 21 May 2021.

[18] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. 2016. Refinement Types for TypeScript. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 310–325. https://doi.org/10.1145/2908080.2908110

[19] WebStorm [n.d.]. WebStorm: The Smartest JavaScript IDE by JetBrains. URL: https://www.jetbrains.com/webstorm. Accessed: 21 May 2021.

[20] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532