Model-Based Testing of Breaking Changes in Node.js Libraries

Anonymous Author(s)

ABSTRACT

Semantic versioning is widely used by library developers to indicate whether updates contain changes that may break existing clients. Especially for dynamic languages like JavaScript, using semantic versioning correctly is known to be difficult, which often causes program failures and makes client developers reluctant to switch to new library versions.

The concept of type regression testing has recently been introduced as an automated mechanism to assist the JavaScript library developers. That mechanism is effective for detecting breaking changes in widely used libraries, but it suffers from scalability limitations that makes it slow and also less useful for libraries that do not have many available clients.

This paper presents a model-based variant of type regression testing. Instead of comparing API models of a library before and after an update, it finds breaking changes by automatically generating tests from a reusable API model. Experiments show that this new approach significantly improves scalability: it runs faster, and it can find breaking changes in more libraries.

1 INTRODUCTION

An important challenge in software maintenance is how library developers can make updates to their libraries without unintentionally breaking the existing clients of the libraries. Library developers commonly use the semantic versioning scheme to indicate if an update contains backward incompatible changes, also called breaking changes. With semantic versioning, updates are marked as major when they are backward incompatible and minor or patch otherwise. Generally, library developers should strive toward creating backward compatible updates since clients often apply such updates automatically, and instant rollout of updates can be critical for security fixes.

A considerable weakness of semantic versioning is that library developers mostly rely on their own estimates when deciding which semantic versioning category an update belongs to. Previous work has shown that developers often incorrectly classify updates as minor or patch despite breaking changes [3, 5, 8, 14, 16]. This is especially problematic for dynamically typed languages, like JavaScript, where mismatches between the library and the client code are not detected until run-time. JavaScript application programmers use libraries extensively; the npm[1] repository contains more than 750 000 modules, mostly libraries, many of which have thousands of daily downloads and are frequently updated.

A few tools exist for helping developers detect breaking changes before an update is released to the clients. Examples include APIDiff, Clirr, and Revapi for Java [7], the `elm diff` tool[2] for elm, and NoRegrets [14] and *dont-break*[3] for JavaScript. A common property of these tools is that they compute the changes to the types of the public API of the library for a given update, and then identify the changes that may break clients. Although this approach can only detect type-related breaking changes, not semantic changes that affect the library functionality but preserve the types, previous work has shown that it is strong enough to catch most breaking changes in practice [14].

The existing techniques NoRegrets and *dont-break* for JavaScript require running the test suites of a library's clients to detect breaking changes when the library has been updated. That approach has several disadvantages. First, installing the client test suites may consume a considerable amount of storage, and running them often takes significant time, although typically only a small part of those test suites is relevant for the library. The *dont-break* tool simply reports breaking changes whenever a client test fails with the updated version of the library. In contrast, NoRegrets uses a technique called type regression testing. It performs a dynamic analysis of the client test executions to infer models of the library API before and after the library update, which leads to more breaking errors being detected and to more actionable error reports for the library developer. However, an important limitation of NoRegrets is that it can only use those clients whose dependencies include the current version of the library. For example, after a new major release of the library, the clients cannot be used by NoRegrets until they have been updated to the new version. Another limitation is that NoRegrets can only use clients that satisfy certain dependency constraints. (We explain this technical limitations of NoRegrets in more detail in Section 7.) As a consequence, we find that NoRegrets does not work well on libraries that only have few available clients.

In this paper, we present a new technique for finding breaking changes in Node.js library updates, which does not suffer from these limitations of existing tools and yet finds more breaking changes. The new technique is implemented in the tool MoTyR.[4] It borrows the concept of dynamically computed API models introduced by NoRegrets, however, MoTyR does not need to re-run all the client tests at every new release candidate of a library. Instead, from a single execution of the client tests it computes an API model that can be used for checking multiple subsequent updates of the library. It does so by using the model to guide a dynamic exploration of the library, while checking that the types of the values that flow between the library and the clients are compatible with the model.

Since MoTyR only uses the client tests to generate the initial model, it avoids running the irrelevant code of the client tests in the checking phase, which makes it considerably faster than NoRegrets. The models are typically not very large, so they are also

[1] `https://www.npmjs.com`

[2] `https://package.elm-lang.org/`

[3] `https://www.npmjs.com/package/dont-break`

[4] **Mo**del-based **T**ype **R**egression tester

ESEC/FSE 2019, 26–30 August, 2019, Tallinn, Estonia
2019. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
`https://doi.org/10.1145/nnnnnnn.nnnnnnn`

more easily stored than the whole set of clients. Additionally, this new approach is less sensitive to the versioning constraints in the client dependencies, which makes it useful even for libraries with relatively few clients.

In summary, this work makes the following contributions:

- We present a new model-based approach to type regression testing, designed to overcome the main practical limitations of the NoRegrets technique.
- We demonstrate by an experimental evaluation of our implementation MoTyR that it is able to find more breaking changes than NoRegrets, an order of magnitude faster and requiring less space, and that MoTyR works better for libraries where relatively few clients are available. Specifically, applying MoTyR to a total of 1 914 minor or patch updates of 25 Node.js libraries with varying numbers of clients detects 84 breaking changes, where NoRegrets in comparison only finds 28.

2 MOTIVATING EXAMPLE

To illustrate the practical limitations of the existing techniques for detecting breaking changes in JavaScript libraries, consider *big-integer*,[5] a library that provides support for arbitrary precision integer arithmetic.

Example 1 The patch update of *big-integer* from version 1.4.6 to version 1.4.7 introduced a new representation of integers that are small enough to fit in a primitive number, based on a new constructor named `SmallInteger`. For example, the library internally uses a function `parseValue` to create a representation of a big integer from some user-supplied input, for example, a string representation of the integer in decimal form. The update contains the following changes to `parseValue`:

```
1  //big-integer 1.4.6
2  function parseValue (v) {
3    ...
4    return new BigInteger(...);
5  }
6  //big-integer 1.4.7
7  function parseValue (v) {
8    if (isPrecise(v)) {
9      return new SmallInteger(v);
10   }
11   ...
12   return new BigInteger(...);
13 }
```

The new `SmallInteger` constructor is used instead of `BigInteger` when the user-supplied value is small enough (lines 8–10). The `SmallInteger` constructor internally uses a primitive number to represent its value, which makes it more efficient than the array of numbers used by `BigInteger`. To make the underlying representation transparent to the users, the update also includes operations on `SmallInteger` objects mirroring the existing functionality of `BigInteger`. All the operations performed on these types are overloaded, for example, it is possible to seamlessly multiply a `SmallInteger` with a `BigInteger`. With this optimization, the *big-integer* library became much faster at processing smaller integers with the release of version 1.4.7.

[5] `https://www.npmjs.com/package/big-integer`

However, the `valueOf` method behaves differently. On `BigInteger` it returns a best-effort conversion to a primitive number, while on `SmallInteger` it instead returns a reference to the `SmallInteger` object itself. Because of this difference, the update contains a breaking change that should not have been introduced in a patch update. The severity of this breaking change is demonstrated by the fact that the *big-integer* developers released a patch of this issue (version 1.4.12) even after version 1.5.0 was released to also accommodate the clients that do not automatically apply minor updates.

As mentioned in Section 1, the *dont-break* tool works by running the test suites of clients of the library before and after the update. One such client is the *deposit-iban*[6] library, which contains the following code:

```
14 const bigInt = require('big-integer');
15 export function isValidIban(iban) {
16   ...
17   const bban = ... // '62000000002021023290006182700';
18   const checkDigitBigInt = bigInt(bban);
19   let checkDigitNumber =
20     String(98 - checkDigitBigInt.mod(bigInt('97')));
21   ...
22 }
```

Before the upgrade of *big-integer*, in line 20 the `mod` method returns a `BigInteger` object whose `valueOf` method is invoked implicitly at the '–' operator. After the upgrade, `mod` instead returns a `SmallInteger` object with the different `valueOf` method, which returns the `SmallInteger` object instead of a primitive number. This means that at the '–' operator, JavaScript implicitly now also invokes `SmallInteger`'s `toString` method, which returns a string that in turn is coerced into a primitive number. The test suite of *deposit-iban* does reach the `isValidIban` function and the different behavior in line 20. Nevertheless, all the tests still succeed with the broken version 1.4.7 of *big-integer* because the JavaScript runtime coerces the result of the `mod` call to the same primitive number as in version 1.4.6, even though the behavior of `valueOf` has changed. As a consequence, *dont-break* misses the breaking change.

In contrast, the NoRegrets tool can detect this breaking change using *deposit-iban*'s test suite. The API model produced by NoRegrets for *big-integer* version 1.4.6 will state that `valueOf` returns a `number`, whereas the model of version 1.4.7 will state that `valueOf` returns an `object`. Clearly, these two types are not interchangeable, so a breaking change is reported. However, NoRegrets still runs all of *deposit-iban*'s test suite, which consists of 45 separate tests where only some use *big-integer*. That test suite was naturally developed to test the logic of *deposit-iban* rather than that of *big-integer*, so even for those tests that do use *big-integer*, most of the work is irrelevant from the perspective of determining whether the API of the *big-integer* library has changed.

With our new approach, MoTyR, the test suites of the clients are still required to infer the initial API model of *big-integer*. However, once this initial model has been constructed, MoTyR checks the types of the library's API by dynamically exploring it based on the information in the model. Specifically, for the aforementioned breaking change, all MoTyR needs to do is to load the *big-integer* library, call the `mod` function with the right arguments, call `valueOf` on the result, and assert that the type is compatible with the type

[6] `https://www.npmjs.com/package/deposit-iban`

in the model. Expressed as JavaScript code, this corresponds to executing the following test:

```
23 const bigInt = require('big-integer');
24 assert(typeof(bigInt('62000000020210232900618 2700')
25                .mod(bigInt('97')).valueOf())
26         === "number")
```

With this approach there is no need for storing the entire *deposit-iban* client and its test suite (and similarly for all the other clients of *big-integer*), and the breaking change detection phase is much faster since the irrelevant work is avoided.

3 OVERVIEW

The purpose of MoTyR is to help Node.js library developers determine if a modification of a library results in breaking changes in the types of the library's API.

The intended usage is as follows. First, the library developer uses the *model generation phase* of MoTyR that automatically fetches publicly available clients and their tests from GitHub, and then runs the tests and simultaneously records the interactions with the library to form a model of the library's API. When the library developer is later ready to release an update, MoTyR is run in the *type regression testing phase*[7] on the updated version of the library code, and a set of non-backward-compatible differences in the API types is reported. If the set is empty, then the library developer can confidently mark the update as either minor or patch, since the API types of the library probably did not change. On the other hand, a nonempty set indicates changes to the API. If a manual inspection of the causes of the warnings produced by MoTyR shows that the differences are unlikely to cause problems in practice, then the developer can go ahead and release the new code as a minor or patch update. If instead the warnings reveal more serious breaking changes, then the developer can either release the changes as a major update (and document the breaking changes accordingly), or, if the changes were unintended, choose to fix the library code and rerun the checking phase of MoTyR to check that the type regressions are gone and that no new type regressions were introduced in the process. The execution of MoTyR's checking phase is fast enough to be integrated into the library's integration test suite, such that MoTyR can be used continuously to check check for type regressions during the development cycle.

Because of the dynamic nature of JavaScript, the API models produced by MoTyR are of course not perfect, so the tool should be used as a supplement, not a substitute for the developer's understanding of the library code. However, as shown in previous work [14] and in the experimental evaluation of MoTyR (Section 6), library developers often overlook breaking changes, and MoTyR can catch many of them.

Example 2 Continuing Example 1, MoTyR will first generate an API model for version 1.4.6 of *big-integer*, by running the test suite of *deposit-iban* while dynamically analyzing the interactions between the client and the library. The main constituent of an API model is a map from *dynamic access paths* to *types*, which we define formally in Section 4. Intuitively, a dynamic access path (or path, for short) refers to the value that appears as result of performing a sequence of operations, for example, a call from the client to a library function, or a write within the library to an object originating from the client. Types include both ordinary JavaScript types, such as `string` and `number`, and concrete primitive values. For example, the following paths expose the problem from Example 1:

$$
\begin{aligned}
p_1 &= \texttt{require}(\textit{big-integer}) \xrightarrow{a} \texttt{ARG}_0 \\
p_2 &= \texttt{require}(\textit{big-integer}) \xrightarrow{b} \texttt{ARG}_0 \\
p_3 &= \texttt{require}(\textit{big-integer})()^b \\
p_4 &= \texttt{require}(\textit{big-integer})()^a.\texttt{mod} \xrightarrow{c} \texttt{ARG}_0 \\
p_5 &= \texttt{require}(\textit{big-integer})()^a.\texttt{mod}()^c.\texttt{valueOf}()^d
\end{aligned}
$$

A model that includes these paths (and many others) is generated when running MoTyR on the client test code shown in lines 14–22. For line 18 when the client calls `bigInt`, the path p_1 models the value being read by the library function when accessing argument number 0, in this case the string `'62000000020210232900618 2700'`. For the second call to `bigInt` in line 20, p_2 similarly refers to the string `'97'`, and p_3 refers to the return value. The path p_4 refers to the value read by the `mod` library function when it reads its argument number 0. Finally, p_5 models the value returned by the implicit call to `valueOf` at the '`-`' operator in line 20 as the type `number`. The labels a, b, and c uniquely identify the function calls involved; specifically, we see that p_1, p_4, and p_5 involve the same call to `require('big-integer')`, and p_4 and p_5 involve the same call to `mod`. An API model additionally contains information about the order by which the paths have been observed and how values flow between paths, which we describe in Section 4.

Such a model contains enough information to enable MoTyR to automatically produce type regression tests like the one shown in lines 23–26. For example, when MoTyR is run in the checking phase on version 1.4.7 of *big-integer*, it simulates the individual actions of the path p_5 and observes that `valueOf` returns an object instead of a number, and therefore issues a type regression warning. To reproduce the actions of p_5, MoTyR obtains arguments for the calls to `mod` and the main function of *big-integer* simply by inspecting the model at p_1, p_2, and p_4. This process of generating tests from the model is described in more detail in Section 5.

4 PHASE I: MODEL GENERATION

Inspired by NoRegrets, we obtain realistic executions of the library of interest by leveraging the publicly available test suites of clients of the library. Running the test suites using program instrumentation with ES6 proxies, MoTyR can monitor the flow of values between the clients and the library, which makes it possible to build a model of the public API of the library. Although this phase of MoTyR is conceptually very close to NoRegrets, for completeness we briefly explain MoTyR's notion of API models, and we point out the important differences.

API models An API *model* in MoTyR is a triple (π, σ, ρ). We begin by explaining the first component, π, which is map of the form $\pi : \mathit{Path} \to \mathit{Type}$ that associates types with elements of a library API much like in NoRegrets. The set *Path* consists of *dynamic access paths*, each being a sequence of *actions*, as described in the following grammar by p and α, respectively.

[7] Using the terminology introduced by Mezzetti et al. [14], a *type regression* is a change in the type signatures of the library API that is incompatible with the mutual expectations of the client and the library developers.

$$p ::= \varepsilon \mid \texttt{require}(n) \mid p\,\alpha$$
$$\alpha ::= .n \mid ()^{\kappa} \mid {}_{\texttt{new}}()^{\kappa} \mid \xrightarrow{\kappa}_{\text{ARG}j} \mid \cdot n \rightarrow$$

Dynamic access paths can be thought of as references to elements of the library's API. Each kind of action corresponds to a JavaScript operation, and a path corresponds to a sequence of operations. All paths begin with a `require`(n) action, where n is the name of a Node.js module.[8] The `require`(n) action can be followed by a sequence of property reads (denoted $.n$ where n is a property name), function applications (denoted $()^{\kappa}$ where κ is explained below) and argument reads (denoted $\xrightarrow{\kappa}_{\text{ARG}j}$ where j indicates the zero-indexed position of the argument). We refer to Mezzetti et al. [14] for further description of the different kinds of actions that also appear in NoRegrets.

Unlike NoRegrets, MoTyR additionally supports a new write action (denoted $\cdot n \rightarrow$, where n is the property being written), which is used to capture side-effects of the client and library functions in the API models. The κ label in the actions, which is also exclusive to MoTyR models, is used to distinguish calls to the same function.[9] In an argument read action, $\xrightarrow{\kappa}_{\text{ARG}j}$, the label κ identifies the function call for which the argument is being read. The purpose of these modifications to the *Path* mechanism becomes clear when we explain the type regression testing phase in Section 5.

As an example, the *qs*[10] library has a method named `parse` that in version 2.2.1 unintentionally writes to the `value` property of the object given as argument (this error is described in more detail in Section 6.2). We can refer to the value being written using the path `require`(qs)`.parse` $\xrightarrow{a}_{\text{ARG}0}$ `.value` $\rightarrow$. This path describes the following actions: load the library using `require('qs')`, invoke its `parse` method (with an argument obtained via another part of the model), and then write to the `value` property of its argument. (The action label a is not relevant in this example.) The position of an action in the path shows whether it appears in client code or in library code: every argument read or write action corresponds to switching side, as indicated by the $\rightarrow$ symbols. For this specific path, invoking `require('qs')` and accessing its `parse` method happens in client code, but reading the method argument and writing to its `value` property happens in library code. We say that a path is *covariant* or *contravariant* if its last action is in client code or library code, corresponding to an even or odd number of $\rightarrow$ symbols, respectively.

A type $t \in \mathit{Type}$ can be a standard JavaScript runtime type (`number`, `boolean`, `object`, etc.), a Node.js specific type like `stream` or `event-emitter`, or the special type $\circ$ that we use in models for paths that are not part of the library's public API.

$$t ::= \circ \mid \texttt{undefined} \mid \texttt{string} \mid \texttt{boolean} \mid \texttt{number} \mid \texttt{object} \mid \texttt{function} \mid \texttt{array} \mid \texttt{set} \mid \texttt{map} \mid \texttt{event-emitter} \mid \texttt{stream} \mid \texttt{throws} \mid \mathit{prim}$$

Unlike in NoRegrets, a type can also be a JavaScript primitive value (here denoted *prim*), similar to how primitive values can be used as types in TypeScript.[11] This extension is made because MoTyR needs to reconstruct arguments for library functions in the type regression testing phase.[12]

A subtyping relation $<:$ is used by MoTyR such that type changes that satisfy the relation are permitted:

$$t <: \circ \qquad \texttt{function} <: \texttt{object} \qquad \cdots$$

The $t <: \circ$ rule, which states that everything is a subtype of $\circ$, is violated when the library reads or writes new properties on client provided objects after an update. Functions are subtypes of objects since JavaScript functions are basically callable objects. Furthermore, the $<:$ relation includes a few rules stating that some of the Node.js specific types are subtypes of `object` and/or `function` (abbreviated by "$\cdots$" above).

The second and third components of the model triple, σ and ρ, are new to MoTyR. The second component, σ, is a partial map $\sigma : \mathit{Path} \hookrightarrow \mathbb{N}$ that associates a unique number with each path p where $\pi(p) \neq \circ$. It has the following property: for any two paths p and p', $\sigma(p) < \sigma(p')$ if and only if p is encountered before p' in the model generation phase described below. This information is needed by the testing phase to be able to invoke the library functions in the same order as the client on which the model is based, which we will later demonstrate in Example 4. For paths that are encountered multiple times during the model generation, we always use the observations from the first one.

The third component, ρ, is a binary relation ρ of the form $\rho \subseteq \mathit{Path} \times \mathit{Path}$. This relation is used to track how values flow from one path into another; for example, if a value returned by a library function call, represented by the path p, is later passed back to the library as an argument to a library function, where the argument is represented by the path p', then $(p, p') \in \rho$.

Model generation To generate an API model (π, σ, ρ) of a given library based on a collection of client test suites, MoTyR instruments the loaded module with ES6 proxies, runs the client test suites, and records the interactions between the library and the clients. The details of how this instrumentation works are explained by Mezzetti et al. [14], except for some straightforward adjustments to accommodate our variant of API models.

One of the adjustments involves extending the π component with a new path p. The type associated with p now depends on the variance of p: if p is contravariant and the value v observed at p is of a primitive type t, then v is used as the type instead of t. For example, if the value is the string `'foo'` and p is contravariant then the type is `'foo'`, otherwise it is `string`. Thereby we ensure that the type regression testing phase of MoTyR has values available for library function arguments, and the model compression mechanism, which we will describe shortly, is not restricted by too specific types.

Another adjustment involves extending the ρ relation whenever a value flows from one path to another. In Example 2, the value created by the `bigInt` call in line 20 represented by the path p_3 flows into argument of the `mod` call represented by the path p_4, resulting in (p_3, p_4) being added to ρ.

[8] Node.js libraries are loaded via the built-in `require` function, as shown in Section 1.
[9] Because of the introduction of the κ labels, MoTyR does not need to track the number of arguments at calls as done by NoRegrets. The array access abstraction, which is used in NoRegrets to model reads of array indices, is also not needed in MoTyR. Instead the property read action $.n$ is used where n is the array position being read.
[10] `https://www.npmjs.com/package/qs`
[11] `https://www.typescriptlang.org/docs/handbook/advanced-types.html`
[12] For readers who are familiar with NoRegrets: MoTyR does not use intersection types nor union types. NoRegrets uses intersection types to model JavaScript prototype chains, however, for MoTyR to reconstruct the client arguments in tests, it must know exactly on which object in a prototype chain a property resides, so extended paths such as p`.prototype` are used instead to refer to the prototype of a path p. Union types are used by NoRegrets to model polymorphic functions, but are not needed in MoTyR since different calls are distinguished using the κ labels.

$p \in Path$	$\pi(p)$	$\sigma(p)$
require(*lib*)	object	1
require(*lib*).f	function	2
require(*lib*).f $\xrightarrow{a}_{\text{ARG0}}$	true	3
require(*lib*).f()^{a}	object	4
require(*lib*).f $\xrightarrow{b}_{\text{ARG0}}$	false	5
require(*lib*).f()^{b}	object	6
require(*lib*).f()^{a}.p	number	7
require(*lib*).g	function	8
require(*lib*).g $\xrightarrow{c}_{\text{ARG0}}$	object	9
require(*lib*).g()^{c}	number	10
other paths	∘	*undefined*

$\rho = \{(\text{require}(lib).\text{f}()^{b}, \text{require}(lib).\text{g} \xrightarrow{c}_{\text{ARG0}})\}$

Figure 1: API model for Example 3.

Example 3 For the following simplistic library and client, MoTyR constructs the model shown in Figure 1.

```
27 //library 'lib'
28 module.exports.f = function (flag) {
29   if (flag) { return { p : 42}; }
30   else { return {}; }
31 }
32
33 module.exports.g = function (o) {
34     return 87;
35 }
36 //client test suite
37 const lib = require('lib');
38 const o1 = lib.f(true);
39 const o2 = lib.f(false);
40 assert(o1.p === 42);
41 assert(lib.g(o2) === 87);
```

The client code loads the library *lib*, calls the `f` method with the argument `true` and stores the result in `o1`, then it calls `f` with the argument `false` and stores the result in `o2`. Finally it checks that `o1.p` is `42` and that `lib.g` called with `o2` as argument is `87`.

The paths and types of every operation taking place at the boundary of the client and the library are recorded in π: the read of the `f`, `p`, and `g` properties, the two calls to `f`, and the call of `g`, and finally the argument reads at the three calls. Notice how the two calls to `f` are distinguished using the labels a and b in the paths. If we were to abstractly refer to both calls using just one path, then there would be no way to determine if the p property should be present on the return value only when `f` is called with the argument `true`, when it is called with the argument `false`, or in both cases. The fact that the argument passed to `g` is the value returned by the call to `f` in line 39 is indicated by the single entry in ρ.

Model compression The action label κ is used to distinguish different calls to the same library function, as mentioned above. Because of these labels, models may become much larger than in NoRegrets if the same library function is called many times. To mitigate this model size explosion problem, we add a simple compression mechanism. The idea is to only include one call of a polymorphic function for each of its possible return types since that suffices for full coverage of the types. We first identify pairs of paths $q = p()^{a}$ and $q' = p()^{a'}$ where q and q' are covariant paths representing two calls to the same function only separated by different labels, $a \neq a'$. If all paths $s = qr$ and $s' = q'r$, where r is a sequence of actions that does not begin with an argument read action, the types are equal, i.e. $\pi(s) = \pi(s')$, and s and s' do not appear in ρ, then we remove q from the model and all paths that have q as a prefix. The reason for skipping paths with an argument read action is that they are only used to synthesize arguments in the type regression testing phase, so covering all argument types does not increase the recall of MoTyR. Paths appearing in ρ are not removed since they may be needed as arguments to other functions.

5 PHASE II: TYPE REGRESSION TESTING

The key novelty of MoTyR is the use of model-based testing, based on the automatically generated models. When the library developer has obtained an API model of one version of a library and later wishes to release an update, MoTyR uses the model to perform a dynamic exploration of the updated library while testing for type regressions relative to the model.

The dynamic exploration consists of two primary steps:

(1) For every covariant path p where $\pi(p) \neq \circ$ type, MoTyR executes the actions described by p and checks that the type of the resulting value is compatible with the type $\pi(p)$.
(2) For any contravariant path p whose actions happen to be executed by library code as a consequence of step 1, MoTyR checks that $\pi(p) \neq \circ$.

Intuitively, the first step corresponds to checking the types of the values that are passed by the library to its client. For example, a library method call that returned a string before the update should not return a number after the update. The second step corresponds to checking that the requirements of the values supplied by the client to the library are not strengthened in the update. For example, after the update, a library function should not read more properties of an object that has been supplied by the client.

If any of the checks performed in these two steps fail, then it is an indicator that the API of the library has changed in a way that could be breaking clients.

API exploration In the type regression testing phase, MoTyR mimics clients by performing the computations corresponding to the actions of the covariant paths in the model. Performing these computations sometimes requires obtaining values from other paths, which is handled by a synthesis procedure described below.

Example 4 To call `g` in line 41 in Example 3, we first need to call `f` in line 39 since its return value is used as argument to `g`.

It is also common for paths to have shared prefixes, for example, all paths in Figure 1 have require(*lib*) as a prefix. If two paths share a common prefix, the value obtained for the prefix should be reused in the checks of both paths.

To accommodate these requirements, MoTyR represents a model π as a tree τ. Every node x in τ is a triple $x = (p_x, C_x, v_x)$ consisting of a path $p_x \in Path$, a set C_x of child nodes, and a JavaScript value v_x that is assigned when x has been processed as explained below. The tree has one node for each path p where $\pi(p) \neq \circ$. A node x is child of x' if $p_x = p_{x'}\alpha$ for some action α. In the exploration of the API, MoTyR traverses τ starting at the root, and when a node x has been processed, the resulting value is stored in the tree as v_x. A child is never processed until its parent has been processed. When MoTyR has to choose between two nodes x and x' to process next, it picks x if $\sigma(p_x) < \sigma(p_{x'})$. Thereby the nodes are processed in the same order as they were added to π in the model generation phase.

In the process of exploring the API, MoTyR needs to convert actions into their corresponding JavaScript operations. To process a node x whose parent is x', MoTyR performs a pattern match of p_x and executes the associated operations:

`require(`n`)`**:** Load the module by calling `require(`n`)`.
$p.n$**:** Fetch the value $v_{x'}$ (corresponding to p) and read its n property.
$p()^\kappa$**:** First, fetch the value $v_{x'}$, which is the function to be called. Next, construct the arguments. Each argument has its own node x^i whose path is $p_{x^i} = p \xrightarrow{\kappa}_{\text{ARG}i}$, which is a child of x'. The argument at position i is constructed by invoking the synthesis procedure described below for the node x^i. Finally, invoke $v_{x'}$ with the synthesized arguments to obtain the result value.
$p_{\mathsf{new}}()^\kappa$**:** Constructor call actions are processed exactly like call actions, apart from the function value being invoked as a constructor (with `new`).
$p \cdot n \rightarrow$**:** Invoke the synthesis procedure for x to produce a value, and then write that value to the property n of $v_{x'}$.

Paths ending in argument read actions are handled by the synthesis procedure described next.

Synthesis of values The synthesis procedure is used above to construct arguments for library function calls and to construct values for writes to library objects. The procedure is invoked with a node x as argument. If there exists a node x' such that $(p_{x'}, p_x) \in \rho$ then the desired value originates from an earlier interaction with the library represented by a path $p_{x'}$, so the value $v_{x'}$ is returned. Otherwise, we proceed according to the type $\pi(p_x)$ of x:

- If the type is a primitive value then that value is returned.
- If the type is `object` or one of the Node.js-specific types, then MoTyR creates a new empty object and wraps it in a proxy object, which is then returned. The purpose of the proxy is twofold. If the proxy is later used as an argument to a function, then that function might read one of its properties, `q`, in which case the proxy looks for a node x' where $p_{x'} = p_x.\mathtt{q}$ among the children of x. If x' is found, then the proxy recursively invokes the synthesis procedure with argument x'. Thereby, the properties of object arguments are constructed by need. If no node is found, then the proxy reports a type regression indicating that the library is now trying to read a property that it did not previously read, cf. step 2. Writes by the library to the proxy are handled similarly to calls from the library to client functions, as described next.
- If the type is `function` then MoTyR creates a new function `f` that behaves as follows when called. If x has a child x' in τ such that $p_{x'} = p_x()^a$, i.e., that path ends in a call action, then a value $v_{x'}$ for x' is obtained by a recursive call to the synthesis procedure. This value is then used as the return value of `f`. Furthermore, the API exploration mechanism described above is invoked recursively on each argument passed to `f`. For each argument at position i, the API exploration checks that it recursively satisfies the type of x^i where $p_{x^i} = p_x \xrightarrow{a}_{\text{ARG}i}$. On the other hand, if no child of x with a call action is found, then MoTyR reports a type regression to indicate that a previously uncalled callback is now being called, cf. step 2. The `f` function is also wrapped in a proxy since functions can also have properties, which may later be read if `f` is used as an argument.

Type checking During the API exploration, MoTyR checks type compatibility of the values obtained for the covariant paths, as mentioned above. If v is the value obtained through the application of the actions of the covariant path p, then v must satisfy $type(v) <: \pi(p)$ where $type(v)$ denotes the type of v. A violation of this property indicates a breaking change in the library's API at p.

6 EVALUATION

As explained in Section 1, the overall goal of MoTyR is to mitigate the scalability issues of NoRegrets. To assess how well MoTyR reaches this goal, we conducted an experiment designed to answer the following research questions.

RQ1 How many breaking changes does MoTyR detect compared to NoRegrets in widely used Node.js libraries, specifically those used in the evaluation of NoRegrets [14, Section 7]?
RQ2 How much faster is MoTyR, and how much space does it require compared to NoRegrets when testing for breaking changes in a library update?
RQ3 Can MoTyR find breaking changes in libraries with fewer clients compared to NoRegrets?

[Our implementation and all experimental data will be submitted for artifact evaluation.]

6.1 Experimental Setup

To answer the research questions, we sampled 25 npm packages from three segments of the npm repository as listed in Table 1. The first five packages are amongst the top 10 most depended upon npm packages and are also the packages used in the evaluation of NoRegrets. Then we have a set of 10 packages sampled around the top 100 most depended upon packages, and a set of 10 packages sampled around the top 1000 most depended upon package. The less depended upon packages have fewer available clients with test suites, which both NoRegrets and MoTyR need for API model generation, but all of the packages are widely used. Most of them have more than 100 000 weekly downloads, so breaking changes in non-major updates can have severe consequences. We skipped packages whose newest version was less than 1.0.0 since such packages are not required to follow semantic versioning. We also skipped very small packages with trivial APIs, such as *is-stream* and *make-dir*, since their update rate is low and their APIs are unlikely to change.

We selected the first major version of every package and applied MoTyR and NoRegrets to every patch and minor update up to the newest version (as of January 2019). For reasons discussed in Section 7, MoTyR is able to use more clients than NoRegrets when generating API models, however, when comparing the execution times of the two tools, we constrained MoTyR to use the same set of clients as NoRegrets to ensure a fair comparison. Because finding clients for many libraries is a time consuming process, we limited the client retrieval phase to consider at most 2 000 packages. For NoRegrets, we built the API model only for the first version, and then reused this model in the test of every update. The execution time of MoTyR is measured as the time it takes to execute the type regression testing phase, whereas for NoRegrets it is the time it takes to generate the post-update model and compare it with the

Table 1: Experimental comparison of MoTyR vs. NoRegrets.

Benchmark				MoTyR				NoRegrets				
Name	**LOC**	**Dependents**	**Updates**	**Clients**	**Model size**	**Coverage**	**BC**	**Clients**	**Client size**	**Coverage**	**BC**	**Speed-up**
debug 2.0.0	154	26 146	19	504	45 kB	50%	1	85	15 709 kB	47%	1	41.70x
async 2.0.0	1 682	26 699	5	398	1 168 kB	61%	7	70	25 732 kB	37%	3	7.54x
lodash 3.0.0	3 962	83 992	16	287	1 469 kB	65%	6	47	17 219 kB	29%	2	1.85x
moment 2.0.0	1 041	28 591	31	273	209 kB	80%	7	4	10 934 kB	44%	2	7.21x
express 3.0.0	1 011	30 561	95	183	1 281 kB	41%	11	5	32 628 kB	43%	18	3.70x
mime 1.0.0	289	3 854	33	23	3 kB	85%	0	4	14 117 kB	38%	1	27.19x
aws-sdk 2.0.1	4 821	9 223	606	27	7 kB	26%	2	2	11 944 kB	20%	0	6.57x
mysql 2.0.0	3 476	4 052	34	111	181 kB	55%	7	2	20 359 kB	49%	0	11.89x
joi 9.0.0	3 724	5 606	35	409	1 421 kB	70%	6	1	94 428 kB	5%	0	25.66x
minimatch 1.0.0	660	3 239	13	415	160 kB	73%	0	15	16 358 kb	43%	0	27.19x
autoprefixer 1.2.0	2 668	5 109	95	64	257 kB	75%	10	1	11 364 kB	77%	1	5.85x
qs 1.0.0	220	4 110	43	237	84 kB	93%	14	0	N/A	N/A	0	N/A
immutable 1.0.0	115	5 326	60	2	3 kB	26%	2	0	N/A	N/A	0	N/A
ora 1.0.0	103	4 623	5	179	19 kB	48%	0	0	N/A	N/A	0	N/A
mongoose 1.0.0	2 105	6 137	474	52	19 kB	30%	2	0	N/A	N/A	0	N/A
big-integer 1.0.0	312	357	89	20	183 kB	67%	1	0	N/A	N/A	0	N/A
boxen 1.0.0	111	634	6	43	15 kB	80%	0	0	N/A	N/A	0	N/A
react-onclickoutside 4.0	69	516	46	11	0 kB	0%	0	0	N/A	N/A	0	N/A
d3-shape 1.0.0	1 528	449	11	38	739 kB	26%	0	0	N/A	N/A	0	N/A
webpack-stream 2.0.0	138	302	14	23	259 kB	70%	1	0	N/A	N/A	0	N/A
qiniu 1.2.0	219	254	38	14	3 kB	28%	1	0	N/A	N/A	0	N/A
koa-send 1.0.0	63	306	22	25	1 kB	24%	2	0	N/A	N/A	0	N/A
twilio 1.0.0	530	506	89	24	45 kB	53%	1	0	N/A	N/A	0	N/A
wreck 2.0.0	413	375	18	44	446 kB	42%	0	0	N/A	N/A	0	N/A
node-rest-client 1.0.0	354	316	17	29	289 kB	39%	3	0	N/A	N/A	0	N/A
Total							84				28	
Arithmetic mean					332 kB	52%			24 617 kB			15x

pre-update model. In both cases, this reflects the work done when testing a new update of a library for breaking changes.

For every type regression reported by the two tools at minor or patch updates, we manually inspected the type regression to identify its cause and determine if it is an actual breaking change (meaning that the type of some element of the library API has changed) or a false positive. It is common for one breaking change to result in multiple type regression warnings, for example, if the return type changes for a function with many call actions then a type regression is reported for every call. Such related regressions are easy to identify by their common structure, so we group them and only count them as one breaking change.

6.2 Results and Discussion

We present the results of running NoRegrets and MoTyR on the 25 benchmarks in Table 1. The columns contain left to right: the benchmark name and the major version on which the testing was started, lines of code in the initial version excluding tests, total number of direct dependents in npm, numbers of minor and patch updates, the number of clients found by the client detection phase of MoTyR, the average MoTyR model size per client, the statement coverage of MoTyR in the initial benchmark version, the number of breaking changes (BC) found by MoTyR, the number of clients found by the client detection phase of NoRegrets, the average NoRegrets client size, the statement coverage of NoRegrets in the initial benchmark version, the number of breaking changes found by NoRegrets, and the average speed-up ratio of MoTyR compared to NoRegrets. For both tools, the reported numbers of breaking changes are only counting true positives, and excluding duplicates with same root cause as explained above.

RQ 1 Looking at the first 11 rows, which are the benchmarks where NoRegrets has a non-empty set of usable clients, we see that MoTyR finds at least as many breaking changes as NoRegrets for all benchmarks apart from *mime* and *express*. In total, MoTyR detects 84 breaking changes, whereas NoRegrets only detects 28.

The breaking changes found by MoTyR include 11 of those found by NoRegrets. There are two reasons why the NoRegrets breaking changes sometimes go undetected by MoTyR. First, the clients used by NoRegrets are not necessarily a subset of the clients used by MoTyR. The reason is that MoTyR will always use the newest version of a client that has the library as a dependency since it is more likely to utilize more of the library than earlier versions, however, for reasons we describe in Section 7, NoRegrets will always pick a version of the client that satisfies the pre-update version constraint. Second, for some benchmarks MoTyR is not able to synthesize values with enough precision to faithfully reconstruct the library-client interaction on which the model is based. In our experiments, this situation occurs because the model generation phase of MoTyR uses ES6 proxies to monitor the interaction between the client and the library, but some values do not tolerate proxification well. For example, `ServerResponse` objects, which are commonly used with the HTTP library of Node.js, will crash Node.js if they are wrapped in proxies at certain places in the HTTP library. Therefore, MoTyR must avoid using proxies on such objects, which means that their exact structure cannot by synthesized by MoTyR in the checking phase, so MoTyR has to use default values instead. This problem is especially prevalent in the *express* benchmark since it uses the HTTP library of Node.js extensively. (With a further implementation effort it might be possible to mitigate such problems; we plan to investigate this in future work.)

In addition to the 84 breaking changes detected by MoTyR, the tool emitted 4 false positives (not shown in the table). False positives may appear due to, for example, the issues with the proxy mechanism described earlier. Some of the correctly detected breaking changes are of course more serious than others; we show some examples as case studies below.

In summary, MoTyR successfully detects more than twice as many breaking changes compared to NoRegrets.

RQ 2 Looking at the speed-up column of Table 1, we see that MoTyR on average runs the type regression testing phase 15 times faster than NoRegrets generates and checks the post-update model. For some libraries, for example the *debug* library, MoTyR is 41.70 times faster than NoRegrets, whereas for *lodash* the speed-up is only 1.85x. The relatively large difference in the speed-ups is explained by various factors, for example, how much irrelevant code (non-library code) is run by the client tests.

The actual time it takes MoTyR and NoRegrets to check an update for type regressions naturally depends on the size of the generated model and complexity of the client test suites. The mean time it takes NoRegrets to check an update on the 11 benchmarks where clients are available is 96 seconds, and the mean time for MoTyR on the same benchmarks is only 15 seconds. Excluding the outliers *lodash* and *async*, MoTyR checks each update in less than 6 seconds.

The numbers also show that MoTyR requires substantially less space than NoRegrets. The average size of a library model produced by MoTyR is 332 kB per client used for the model construction, whereas NoRegrets requires almost 75x more space.

In summary, MoTyR is more than an order of magnitude faster than NoRegrets when testing a library update for breaking changes, and it requires substantially less space to run. This means that MoTyR can appropriately be included as part of a library's integration test suite, thereby aiding developers during a development cycle of an update.

RQ 3 For the second segment of the benchmarks (i.e., the libraries sampled around top 100), MoTyR finds breaking changes in 7 out of 10 benchmarks, and for the third segment (i.e., the libraries sampled around top 1000) it finds breaking changes in 6 out of 10 benchmarks. In comparison, NoRegrets only finds breaking changes in 2 of the benchmarks from the second segment and in none of the benchmarks in the third segment. This shows that MoTyR scales much better than NoRegrets to libraries with fewer clients.

For most libraries where MoTyR finds no type regressions, the generated tests cover on average 50% of the statements in the library, which provides some indication that those updates are in fact non-breaking. One exception is *react-onclickoutside* where all of the models generated by MoTyR are empty. That package is a plugin for the browser UI library *react*, which means that it is unlikely that any clients have automated tests that use it.

Case studies To give the some insight into the nature of the breaking changes that MoTyR can detect, we describe some representative examples.

Example 5 The *qs* package is a library for parsing query strings. As an example, `qs.parse("p=foo")` returns the object `{p : "foo"}`. A special feature of the package is that it supports parsing of objects where some on the properties are query strings that are parsed recursively, for example, `qs.parse({'a[b]': 'c'})` returns the nested objects `{a : {b : "c"}}`.

In the update of *qs* to version 2.2.1, a mistake was introduced that resulted in the `parse` function sometimes overwriting existing properties on object arguments. This mistake is revealed by MoTyR through a type regression on the path

$$p = \texttt{require}(qs).\texttt{parse} \xrightarrow{a}_{\text{ARG}_0} \cdot\texttt{value} \longrightarrow$$

where $\pi(p) = \circ$ but a value of type `string` is written in version 2.2.1. For most cases this overwrite in benign because `parse` overwrites the property with its existing value, however, specifically for buffers, `parse` writes the result of calling `toString` on the buffer. The following code shows the different behavior:

```
42 const buf = Buffer.alloc(10, 1);
43 var o = { p : buf };
44 qs.parse(o);
45 assert(o.p === buf) // OK in 2.2.0 but error in 2.2.1
```

While the type regression on p does not point directly to this issue, it does indicate that a potential dangerous write is happening on the first argument of `parse`. Even with our limited knowledge of the internals of the *qs* library, we were able to quickly identify the issue with buffers by inspecting the source diff between version 2.2.0 and 2.2.1.

A well-known problem with semantic versioning is that it requires a specification of the library's API,[13] typically in the form of documentation, such that a client knows exactly what the library expects and what it produces, and this is often an unrealistic requirement [1]. Without such a specification, any change to the library that breaks a client might as well be classified as the client not using the library as the library developer intended. With MoTyR, we assume that the clients used in the model generation phase adhere to the library's specification. For clients where this is not the case, MoTyR may produce type regressions, which the library developer could rightly classify as caused by incorrect usage of the library. Nevertheless, we still believe that the library developer can benefit from warnings of that kind, since they may point to ambiguities and underspecified points in the documentation; each such warning reveals that a client developer has misunderstood the specification.

Example 6 The *async* package is a widely used library that provides a large set of utility functions for working with asynchronous functions. One of these functions is `each`, which takes a collection (typically an array), an asynchronous iterator function, and a callback function as arguments. It then asynchronously runs the iterator on every element in the collection, eventually calling the callback when the iteration is done. The iterator function also takes a callback, which it can call with any argument to signal an error, which in turn calls the callback of `each` with the error value. A typical use of the `each` function is demonstrated by the following

[13] *"Software using Semantic Versioning MUST declare a public API"* — rule 1 of the SemVer specification, `https://semver.org/`.

example where the client asynchronously performs some computation on an array of files.

```
async.each(['file1.txt', 'file2.txt'],
  function(file, cb) {
    var err = ... //async operation
    if (err) { cb(err) }
    cb();
  },
  function(err) {
    console.log("error processing files");
  });
```

In the update of *async* from version 2.0.0 to 2.0.1, the `each` function was changed slightly to improve its performance when the collection is an array. While this update is non-breaking when the iterator function is asynchronous, it unfortunately changed the behavior of `each` when the iterator is synchronous. In version 2.0.0 when a synchronous iterator function calls its callback with an error value, the `each` call is directly terminated potentially leaving some elements in the array uniterated. However, in version 2.0.1 the iteration is not terminated on an error, so all elements will always be processed. This breaking change is detected by MoTyR as a type regression on the path

$$\texttt{require}(\textit{async}).\texttt{each} \xrightarrow{a}_{\texttt{ARG}0}.1$$

which refers to the element at index 1 in the array passed to the `each` call. The model states that this path is not read, nevertheless, MoTyR detects a read of this path in version 2.0.0 resulting in a type regression being reported. Upon inspection of this type regression, we find that the iterator function fails when processing the first element of the array, but that does not stop `each` from also beginning the processing of the second element and thereby causing the unexpected read.

Notice that this can only break clients that use `each` with synchronous functions, which is not allowed according to the *async* specification. However, due to either a misunderstanding of the specification or a general lack of knowledge of how asynchrony works in Node.js, many clients use *async* with synchronous functions. A search for "*RangeError: Maximum call stack size exceeded*", which is an error caused by the incorrect use of synchronous functions, on *async*'s issue tracker results in no less than 22 results. Furthermore, the first point in the "Common pitfalls" section of the *async* documentation page mentions the use of synchronous functions as a pitfall.[14]

This example demonstrates that library developers may benefit from warnings reported by MoTyR even in situations where the changed library behavior is intended by the library developer, because many clients fail to follow the library specification and are thereby affected by the change.

While most type regressions reported by MoTyR are true positives, some of them are unlikely to cause problems in practice if the library developer is cautious, as demonstrated by the following example.

Example 7 The *joi* package is a schema validation library, which can be used to validate that objects and strings have a certain structure. Specifically, *joi* has a method `uri` that returns a schema object for validating that strings are valid RFC3986 URIs. The `uri` method takes a configuration object argument, specifying for example that only URIs of certain schemes are allowed:

```
var v = joi.string().uri({ scheme : 'http'})
v.validate('http://foo.bar').error // => null
v.validate('https://foo.bar').error // => ValidationError
```

In version 13.5.0, a new optional property `allowQuerySquareBrackets` was introduced. Setting this property to `true` configures the schema object such that URIs with square brackets in query variables are allowed. MoTyR reports a type regressions for this change, because *joi* reads the path

$$\texttt{require}(\textit{joi}).\texttt{string}()^{a}.\texttt{uri} \xrightarrow{b}_{\texttt{ARG}0}.\texttt{allowQuerySquareBrackets}$$

in version 13.5.0, although the model states that no read should occur on it. However, the developers of *joi* were careful enough to introduce this change such that no existing clients were impacted. If a client does not supply the `allowQuerySquareBrackets` property, then *joi* will automatically assume it is `false` to preserve the old behavior for existing clients. This means that although the library API has changed in a way that could in principle break clients, the type regression is most likely benign.

Even for type regressions that are benign as in Example 7, the library developers may benefit from the warnings provided by MoTyR. The warnings point the library developers to parts of the API where extra care must be taken to ensure backward compatibility and communicate to the client developers that the newly added properties like `allowQuerySquareBrackets` may conflict with existing properties in the client objects.

7 RELATED WORK

Our approach is evidently closely related to the recent work by Mezzetti et al. [14], but the challenge of detecting breaking changes in libraries also appears with other programming languages, and there are also connections to other testing techniques, in particular model-based testing.

Studies of breaking changes in library updates Breaking changes are common across languages and ecosystems [14, 16, 18]. According to Mezzetti et al. [14], at least 5% of JavaScript packages they studied have experienced a breaking change in a non-major update, and that the majority of the breaking changes are due to type-related issues. Brito et al. [3] conducted a study on why and how Java developers intentionally break APIs, concluding that the primary reasons are to add new features (32%), simplify the API (29%), and improve maintainability (24%). Zerouali et al. [19] showed that using strict version number constraint results in slow adoption of security critical updates. Many developers want to adopt semantic versioning, but do not trust that their dependencies adhere to the guidelines [1].

The study by Gyori et al. [6] used client test suites to detect breaking changes in library updates, similar to the *dont-break* methodology mentioned in Section 1 but for Java. They note that it is common practice in industry to use this form of testing, but also that applying certain test case selection criteria could yield a considerable speed-up while preserving coverage, similar to how MoTyR avoids running all the client test suites at every library update.

[14] `https://caolan.github.io/async/`

Tools for JavaScript To our knowledge, only the two tools NoRegrets and *dont-break* exist for detecting breaking changes in JavaScript libraries; the relations between MoTyR and those tools are explained in detail in the preceding sections.

Like MoTyR, NoRegrets also looks for type regressions in Node.js library updates, but it instead generates models for both the pre-update and the post-update version of the library, and then compares the models to identify type regressions [14]. Because NoRegrets needs to compute a model twice to check an update, it is important that the clients' dependency constraints on the library is within the same major number as the pre-update version of the library. As an example of why this constraint is important, consider the case where a library l is at version 2.0.0 and the library developer wants to check some changes for type regressions before releasing 2.1.0. If NoRegrets now picks a client c that depends specifically on version 1.0.0 of l, then c expects l to expose an API that might be considerably different from the API in 2.0.0. If l deprecated a function g in version 2.0.0 and the developer now plans to remove it entirely in version 2.1.0, then if the client uses this function, it will crash with version 2.1.0, which will result in a quite different model and therefore also many type regressions. One of these type regressions will correctly state that g went from `function` to `undefined`, but the rest of them are false positives caused by the premature termination of the client. In contrast, MoTyR is not limited by this constraint since it is able to continue testing of the library even if a type regression that would have crashed the client is detected. This difference allows MoTyR to use a larger set of clients and thereby produce better API models.

Tools for other programming languages For other languages than JavaScript, there are numerous tools that help library developers detect breaking changes. Common to all these tools is that they work for statically typed languages and rely on explicitly typed library APIs, which make it much easier to detect type-related breaking changes than for dynamically typed languages. For Java there is APIDiff [2], Clirr,[15] japicmp,[16] SigTest,[17] and Revapi.[18] The Elm package manager (elm-package) promises to automatically enforce semantic versioning, although being based on changes in the public API of a library, it is limited to detecting type-related breaking changes.

For a dynamically typed language like JavaScript, the public API of a library is not easily identifiable statically, which is why we resort to the use of dynamic analysis for the model generation phase. JavaScript library developers can choose to write TypeScript declaration that define the public API of their libraries. However, declaration files are often full of errors and rarely kept up-to-date making them unsuitable for breaking change detection [11].

A problem related to breaking change detection is how to update clients when their libraries evolve, also called collateral evolution. As an example, the Coccinelle tool [15] has been designed to support Linux developers in this respect, but it does not help the developers determine if and where breaking changes are introduced.

[15] `http://clirr.sourceforge.net/`
[16] `http://siom79.github.io/japicmp/`
[17] `http://wiki.netbeans.org/SigTest`
[18] `https://revapi.org/`

Model-based testing and related techniques Our approach can be seen as a form of model-based testing [17]. Models used in model-based testing are typically specified manually, whereas MoTyR's models are inferred automatically based on dynamic analysis of client usage. Example of other testing techniques that apply similar ideas have been proposed by Joshi and Orso [9], Krikava and Vitek [10], McCamant and Ernst [12, 13], and Elbaum et al. [4].

The SCARPE tool by Joshi and Orso [9] uses a capture phase that generates a model of a software component based on live executions, and a replay phase that can produce regression tests from the model. This is reminiscent of how Krikava and Vitek [10] produce tests for CRAN packages written in R, based on executing the small snippets of executable example code that is often included in the documentation of such packages. In comparison with those techniques, we use the test suites of the client packages to obtain realistic executions of the library.

The techniques by McCamant and Ernst [12, 13] construct logical models of software components written in C by dynamically inferring likely invariants. Incompatibilities at component upgrades can then be detected by comparing the models using an automatic theorem prover.

The idea of extracting new tests from existing tests also appears in the test carving technique by Elbaum et al. [4], which aims to generate effective differential unit tests from existing system tests. In comparison, MoTyR exploits the fact that the test suite of a client of a library often indirectly functions as a system test of the library, which makes it possible to generate useful regression tests from existing client test suites.

8 CONCLUSION

Breaking changes in libraries are a major concern for JavaScript developers. For Java, the static type checker helps detecting such issues when building an application with a new version of a library, but due to the dynamic nature of JavaScript, breaking changes are rarely discovered before failures appear at run-time. Type regressions are a kind of breaking changes that manifest as incompatible changes in the types of the method parameters, return values, and object properties that constitute the API of a library. Previous work has shown that type regressions account for many breaking changes in widely used JavaScript libraries. In previous work by Mezzetti et al., the NoRegrets tool introduced the concept of type regression testing for detecting such issues automatically, but it is inefficient and inadequate for libraries with relatively few clients.

By taking a model-based testing approach, our tool MoTyR creates tests from a model of the library API, all fully automatically. As shown in our experimental evaluation, this new approach is significantly more efficient and capable of finding many more breaking changes, especially in libraries with fewer available clients.

With such tool support, it is our hope that JavaScript library developers can make more informed decisions when releasing updates and using semantic versioning. The experiments have also demonstrated that there is room for improvement of the technique, especially concerning the use of proxies in the model generation phase, which we plan to pursue in future work.

REFERENCES

[1] Christopher Bogart, Christian Kästner, and James D. Herbsleb. 2015. When It Breaks, It Breaks: How Ecosystem Developers Reason about the Stability of Dependencies. In *30th IEEE/ACM International Conference on Automated Software Engineering Workshops, ASE Workshops 2015, Lincoln, NE, USA, November 9-13, 2015*. IEEE Computer Society, 86–89.

[2] Aline Brito, Laerte Xavier, André C. Hora, and Marco Tulio Valente. 2018. APIDiff: Detecting API breaking changes. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*. IEEE Computer Society, 507–511.

[3] Aline Brito, Laerte Xavier, André C. Hora, and Marco Tulio Valente. 2018. Why and how Java developers break APIs. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*. IEEE Computer Society, 255–265.

[4] Sebastian G. Elbaum, Hui Nee Chin, Matthew B. Dwyer, and Jonathan Dokulil. 2006. Carving differential unit test cases from system test cases. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, Portland, Oregon, USA, November 5-11, 2006*. ACM, 253–264.

[5] Darius Foo, Hendy Chua, Jason Yeo, Ming Yi Ang, and Asankhaya Sharma. 2018. Efficient static checking of library updates. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. ACM, 791–796.

[6] Alex Gyori, Owolabi Legunsen, Farah Hariri, and Darko Marinov. 2018. Evaluating Regression Test Selection Opportunities in a Very Large Open-Source Ecosystem. In *29th IEEE International Symposium on Software Reliability Engineering, ISSRE 2018, Memphis, TN, USA, October 15-18, 2018*. IEEE, 112–122.

[7] Kamil Jezek and Jens Dietrich. 2017. API Evolution and Compatibility: A Data Corpus and Tool Evaluation. *Journal of Object Technology* 16, 4 (2017), 2:1–23.

[8] Kamil Jezek, Jens Dietrich, and Premek Brada. 2015. How Java APIs break - An empirical study. *Information & Software Technology* 65 (2015), 129–146.

[9] Shrinivas Joshi and Alessandro Orso. 2007. SCARPE: A Technique and Tool for Selective Capture and Replay of Program Executions. In *23rd IEEE International Conference on Software Maintenance (ICSM 2007), October 2-5, 2007, Paris, France*. IEEE, 234–243.

[10] Filip Krikava and Jan Vitek. 2018. Tests from traces: automated unit test extraction for R. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*. ACM, 232–241.

[11] Erik Krogh Kristensen and Anders Møller. 2017. Type test scripts for TypeScript testing. *PACMPL* 1, OOPSLA (2017), 90:1–90:25.

[12] Stephen McCamant and Michael D. Ernst. 2003. Predicting problems caused by component upgrades. In *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference, ESEC/FSE 2003, Helsinki, Finland, September 1-5, 2003*. ACM, 287–296.

[13] Stephen McCamant and Michael D. Ernst. 2004. Early Identification of Incompatibilities in Multi-component Upgrades. In *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings (Lecture Notes in Computer Science)*, Vol. 3086. Springer, 440–464.

[14] Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. 2018. Type Regression Testing to Detect Breaking Changes in Node.js Libraries. In *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands (LIPIcs)*, Vol. 109. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 7:1–7:24.

[15] Yoann Padioleau, Julia L. Lawall, René Rydhof Hansen, and Gilles Muller. 2008. Documenting and automating collateral evolutions in Linux device drivers. In *Proceedings of the 2008 EuroSys Conference, Glasgow, Scotland, UK, April 1-4, 2008*, Joseph S. Sventek and Steven Hand (Eds.). ACM, 247–260.

[16] Steven Raemaekers, Arie van Deursen, and Joost Visser. 2017. Semantic versioning and impact of breaking changes in the Maven repository. *Journal of Systems and Software* 129 (2017), 140–158.

[17] Mark Utting, Alexander Pretschner, and Bruno Legeard. 2012. A taxonomy of model-based testing approaches. *Softw. Test., Verif. Reliab.* 22, 5 (2012), 297–312.

[18] Laerte Xavier, Aline Brito, André C. Hora, and Marco Tulio Valente. 2017. Historical and impact analysis of API breaking changes: A large-scale study. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*. IEEE Computer Society, 138–147.

[19] Ahmed Zerouali, Eleni Constantinou, Tom Mens, Gregorio Robles, and Jesús M. González-Barahona. 2018. An Empirical Analysis of Technical Lag in npm Package Dependencies. In *New Opportunities for Software Reuse - 17th International Conference, ICSR 2018, Madrid, Spain, May 21-23, 2018, Proceedings (Lecture Notes in Computer Science)*, Vol. 10826. Springer, 95–110.