Semantic Patches for Adaptation of JavaScript Programs to Evolving Libraries

Benjamin Barslev Nielsen
Aarhus University
barslev@cs.au.dk

Martin Toldam Torp
Aarhus University
torp@cs.au.dk

Anders Møller
Aarhus University
amoeller@cs.au.dk

***Abstract*—JavaScript libraries are often updated and sometimes breaking changes are introduced in the process, resulting in the client developers having to adapt their code to the changes. In addition to locating the affected parts of their code, the client developers must apply suitable patches, which is a tedious, error-prone, and entirely manual process.**

To reduce the manual effort, we present JSFIX. Given a collection of semantic patches, which are formalized descriptions of the breaking changes, the tool detects the locations affected by breaking changes and then transforms those parts of the code to become compatible with the new library version. JSFIX relies on an existing static analysis to approximate the set of affected locations, and an interactive process where the user answers questions about the client code to filter away false positives.

An evaluation involving 12 popular JavaScript libraries and 203 clients shows that our notion of semantic patches can accurately express most of the breaking changes that occur in practice, and that JSFIX can successfully adapt most of the clients to the changes. In particular, 31 clients have accepted pull requests made by JSFIX, indicating that the code quality is good enough for practical usage. It takes JSFIX only a few seconds to patch, on average, 3.8 source locations affected by breaking changes in each client, with only 2.7 questions to the user, which suggests that the approach can significantly reduce the manual effort required when adapting JavaScript programs to evolving libraries.

I. Introduction

The JavaScript-based npm ecosystem consists of more than a million packages, most of them libraries used by many JavaScript applications. Libraries constantly evolve, and client developers want to use the latest versions to get new features and bug fixes. However, not all library updates are backwards compatible, so the client developers may be discouraged from switching to newer versions of the libraries because of changes that break the client code. Currently, to update a client to a new major version of a library, the client developer needs to examine the changelog to discover which parts of the client code are affected by breaking changes and find out how to adapt the code accordingly – a process which is entirely manual, error-prone, and time-consuming. Existing tools, such as GitHub's Dependabot,[1] only warn about outdated dependencies and provide no other assistance in this process.

A detection pattern language and an accompanying analysis, TAPIR, have recently been proposed for finding locations in client code affected by breaking changes [1]. Inspired by the Coccinelle tool [2, 3], in this paper we build on top of TAPIR and introduce a notion of code templates for expressing how to also patch the affected locations. Paired with a detection pattern, a code template forms a *semantic patch*. Provided with a collection of semantic patches specifying where breaking changes occur (the detection patterns) and how to adapt the affected code (the code templates) for a library update, our tool JSFIX can semi-automatically adapt clients to become compatible with the new version of the library. A run of JSFIX goes through three phases: an analysis phase (based on the TAPIR analysis) for over-approximating the set of affected locations, an interactive phase for filtering away false positives from that set, and a transformation phase for adapting the client code using the code templates. In the interactive phase, JSFIX asks the user a set of yes/no questions about the behavior of the client code, to remedy inherent limitations in the TAPIR analysis; all those questions concern properties the client developer would have to consider anyway if performing the patching manually.

We envision that semantic patches can be written either by the library developer or by someone familiar with the library code, along with the customary changelogs. With JSFIX, all the clients of the library then benefit from the mostly automatic adaptation of their code (and the client developers do not need to understand the notation for semantic patches).

To summarize, the contributions of this paper are:

- We propose a notion of code templates to formalize the transformations required to adapt client code to typical breaking changes. A TAPIR detection pattern together with a code template form a semantic patch. We present the JSFIX tool, which, based on a collection of semantic patches, semi-automatically adapts client code to breaking changes in a library update (Sections III and IV).
- We propose an interactive mechanism for filtering away false positives from the detection pattern matches reported by TAPIR (Section V).
- We present the results of an evaluation based on 12 major updates of popular npm packages and 203 clients, showing that most breaking changes can easily be expressed as semantic patches, and that JSFIX in most cases succeeds in making the clients compatible with the new versions of the libraries. Furthermore, the evaluation demonstrates the practicality of JSFIX. It takes only a few seconds to patch,

[1]https://dependabot.com/

```
 1  -  import { Observable } from 'rxjs/Observable';
 2  -  import { Subject } from 'rxjs/Subject';
 3  -  import 'rxjs/add/observable/timer';
 4  -  import 'rxjs/add/operator/takeUntil';
 5  +  import { defaultIfEmpty, take, takeUntil, tap }
 6  +         from 'rxjs/operators';
 7  +  import { timer, Subject } from 'rxjs';

 8  -  const c$ = (new Subject()).take(1);
 9  -  Observable.timer(warnTimeout)
10  -    .takeUntil(c$.defaultIfEmpty(true))
11  -    .do(() => {...})
12  -    .subscribe();
13  +  const c$ = (new Subject()).pipe(take(1));
14  +  timer(warnTimeout).pipe(
15  +    takeUntil(c$.pipe(defaultIfEmpty(true))),
16  +    tap(() => {...})
17  +  ).subscribe();
```

Fig. 1: Excerpts from *redux-logic* before (marked with red background and '-') and after (green and '+') adapting to the breaking changes in the library *rxjs* 6.0.0.

on average, 3.8 affected locations per client, with only 2.7 questions to the user, and 31 pull requests based on the output from JSFIX have been accepted, which shows that the quality of the patches is good enough for practical use (Section VI).

II. Motivating Example

The *rxjs* library,[2] with more than 17 million weekly downloads, is a popular library for writing reactive JavaScript applications. In April 2018, *rxjs* was updated to version 6.0.0, a massive major update introducing many new features, bug fixes, and performance improvements, but unfortunately also many breaking changes. Our manual investigation of the *rxjs* changelog shows that it contains at least 38 separate breaking changes, many of which involve multiple functions and modules. The developers of *rxjs*, who were probably well aware that these breaking changes would discourage many client developers from upgrading, decided to create both an auxiliary compatibility package that introduces temporary workarounds and a migration guide detailing how clients should adapt to all the breaking changes in the new version of *rxjs*.

While the migration guide is quite helpful (and also not something provided with most major updates of other libraries), it may still take a significant amount of work for a client developer to upgrade to *rxjs* 6.0.0. Consider, for example, the *redux-logic* package that depends on *rxjs*.[3] In September 2018, *redux-logic* was updated to depend on *rxjs* 6.0.0. This update required 784 additions and 308 deletions to 21 files over 3 commits,[4] by no means a small task.

Figure 1 shows two excerpts of the update of *redux-logic* to *rxjs* 6.0.0 (modulo some newlines and insignificant differences in variable naming). The first change is that `Observable` is no longer imported from `'rxjs/Observable'` as in line 1, in fact it is not imported at all. This is because the `timer` function (`Observable.timer` in line 9) in rxjs 6.0.0 should be accessed directly from `'rxjs'` as can be seen by the import in line 7. Therefore line 14 is also updated to use `timer` directly. The second change is that `Subject` should be imported from `'rxjs'` instead of `'rxjs/Subject'`, which is why the import in line 2 is replaced with the import of `Subject` in line 7. The imports in lines 3–4 add properties to `Observable` and rxjs observables (through `Observable.prototype`), but have been removed in the new version. Instead of using those properties, the functions should now be imported from either `'rxjs'` or `'rxjs/operators'` as can be seen in the imports on lines 5 and 7. Line 8 used one of these properties, `take`, which in the new version should be replaced with a call to `.pipe`, where the operator function (`take`) is then provided as an argument, as shown in line 13 in the patched code. For the same reason, lines 9–12 have been updated to use `.pipe` in lines 14–17. Evidently, adapting client code to breaking changes in a library can be difficult and time-consuming, so tool support is desirable.

While the *redux-logic* example is one of the more extreme cases, it clearly demonstrates that updating dependencies is no minor undertaking. Considering that the average npm package already in 2015 had an average of 5 direct dependencies and that number has been growing over time [4], keeping everything up-to-date becomes insurmountable.

Using JSFIX it would have been possible for the *redux-logic* developer to adapt the client code almost automatically. Given a collection of semantic patches that describe the breaking changes in the library, JSFIX is designed to both find the locations in the client code that are affected by the breaking changes, and to adapt those parts of the client code to the new version of the library. The analysis that finds the affected locations is designed such that it leans towards over-approximating, meaning that it may flag too many source code locations as potentially requiring changes, but rarely too few. When it cannot establish with complete certainty whether some source location is affected by the breaking changes, it asks the client developer for advice. In this specific case, the *redux-logic* developer would only have to answer 14 simple yes/no questions, which all concern only *redux-logic* (not *rxjs*).

For transforming the excerpts shown in Figure 1, JSFIX does not ask any questions. However, suppose the analysis were too imprecise to determine that `c$` on line 10 is an rxjs observable, then JSFIX would have asked this question:

src/createLogicAction$.js, 10:14:10:36:
Is the receiver an rxjs observable?

All the 14 questions are of this kind but with different source code locations, and they all originate from such analysis imprecision. With the help from the *redux-logic* developer, the uncertainty can be resolved, and the patches produced by JSFIX for the affected locations successfully adapt the *redux-logic* source code to the new version of the *rxjs* library.

Comparing the JSFIX autogenerated transformations with the patches made manually by the developer of *redux-logic* shows that the transformations are identical (ignoring white-space and the order of property names in imports).

[2]https://www.npmjs.com/package/rxjs
[3]https://github.com/jeffbski/redux-logic
[4]Counted using Git's notion of additions and deletions.

III. Overview

The tool JSFIX is designed to adapt client code to breaking changes in libraries. An execution of JSFIX is divided into three phases: (1) an analysis phase, (2) an interactive phase, and (3) a code transformation phase. As input it takes a client that depends on an old version of a library, together with a collection of *semantic patches* that describe the breaking changes in the library. Each semantic patch contains a *detection pattern* that describes where a breaking change occurs in the library API and a *code template* that describes how to adapt the client code. We explain the notion of semantic patches in more detail in Section IV. As output, JSFIX produces a transformed version of the client that, under certain assumptions described in Section V, preserves the semantics of the old client code but now uses the new version of the library.

The analysis phase uses the TAPIR [1] light-weight static analysis to detect locations in client code that may be affected by breaking changes in the library. We treat TAPIR as a black-box component as it is only loosely coupled with the other phases of JSFIX. The input to TAPIR consists of the client code and the detection patterns coming from the semantic patches, and as output it produces a set of locations in the client code that match the detection patterns, meaning that they may be affected by the breaking changes in the library.

Being fully automatic, TAPIR cannot always find the exact set of affected locations, but the analysis is designed such that it leans towards over-approximating, meaning that it sometimes reports too many locations but rarely too few. Moreover, it is capable of classifying each match being reported as either a high or a low confidence match. In practice, all false positives appear among the low confidence matches, meaning that only those need to be manually validated. In JSFIX, we take advantage of that confidence information. In the second phase of JSFIX, it asks the user for help at each low confidence match, such that the false positives from TAPIR can be eliminated. The text for the questions to the user comes from the semantic patches. The questions all take yes/no answers, and they concern only the client code, not the library code. We describe the interactive phase in more detail in Section V where we also give additional representative examples of questions presented to the user.

Next, JSFIX runs a transformation phase where the client code is patched to adapt to the changes in the library. The transformations are specified using a form of code templates that specify how each affected location should be transformed to become compatible with the new version of the library. The transformation process is explained together with the notation for semantic patches in the next section.

IV. A Semantic Patch Language

To adapt client code to breaking changes in a library, the parts of the client code that use the affected parts of the library API must be transformed accordingly.

Example 1 Lines 18–19 in the following program use the `max` function from the *lodash* library.

```
18    var _ = require('lodash');
19  - _.max(coll, iteratee, thisArg);
20  + _.max(coll, iteratee.bind(thisArg));
```

The optional third argument on line 19, `thisArg`, lets the client specify a custom receiver of the second argument, `iteratee`. In version 4.0.0 of *lodash*, the support for the third argument was removed from 64 functions, including the `max` function. To restore the old behavior, clients using `max` or one of the other 63 functions would have to explicitly bind `thisArg` to `iteratee`. For example, for the program above, line 19 has to be transformed by inserting a call to `bind` as shown on line 20.

Example 2 Lines 21–23 in the program below use the *async* library's `queue` data structure, which holds a queue of tasks (asynchronous functions) to be processed.

```
21    var async = require('async');
22    var q = async.queue(...);
23  - q.drain = () => console.log('Done');
24  + q.drain(() => console.log('Done'));
```

On line 23, a function is written to the `drain` property of the queue. In version 2 of *async*, this function is called when all tasks in the queue have been processed. However, in version 3 of *async*, `drain` is no longer a property the client should write, but instead a function the client should call. The function to be called once the queue has been processed is then passed as an argument to `drain`. Hence, the call to `drain` must be transformed as shown on line 24.

Example 3 Lines 25–26 below import the `find` and `map` functions from the *rxjs* library.

```
25  - import find from 'rxjs/operator/find'
26  - import map from 'rxjs/operator/map'
27  + import {find, map} from 'rxjs/operators'
```

In version 6 of *rxjs*, these import paths `'rxjs/operator/find'` and `'rxjs/operator/map'` are no longer available. Instead, clients must import the functions from `'rxjs/operators'` as demonstrated by the transformed import on line 27.

To automate the transformation of the client code, we define a suitable notion of semantic patches. A *semantic patch*, $\rho \rightsquigarrow \alpha$, models a breaking change and consists of a *detection pattern* ρ that identifies the affected part of the library API (the affected location), and a *code template* α that describes how client code that uses that part of the API can be transformed to adapt to the new version of the library. A semantic patch can also contain question text for the interactive phase, which we describe in Section V. The detection patterns are identical to the patterns used by the API access point detection tool TAPIR, so we omit a detailed description of the pattern language in this paper. Although we treat the accompanying algorithm that performs the matching between the patterns and the client code as a black box, as mentioned in Section III, we provide intuitive explanations of the meaning of the concrete detection patterns that appear in examples in the remainder of this paper.

To adapt a client that uses some library with breaking changes, for example, to perform the transformations in Examples 1–3, we need a mechanism for specifying the

$$
\begin{array}{rcl}
\alpha \in \textit{Expression} & ::= & \ldots \\
 & | & \$\ \textit{RefElement}\ (\texttt{:}\ \textit{RefElement})^{*} \\
 & | & <\ \textit{ModuleElement}^{+}\ > \\
 & | & \#\ i\ \textit{Replacer}^{?} \\
\textit{RefElement} & ::= & \texttt{prop}\ \textit{Replacer}^{?}\ \mid\ \texttt{value}\ \mid\ \texttt{base} \\
 & | & \texttt{callee}\ \mid\ i\ \mid\ \texttt{args}\ \textit{Selector}^{?} \\
\textit{Replacer} & ::= & [\ s_1 \Rightarrow s'_1,\ \ldots,\ s_n \Rightarrow s'_n\ ] \\
\textit{Selector} & ::= & [\ j, k\ ]\ \mid\ [\ j\ ,] \\
\textit{ModuleElement} & ::= & s\ \mid\ /\ \mid\ \#\ i\ \textit{Replacer}^{?}
\end{array}
$$

Fig. 2: Grammar for code templates. The '...' in the first production refers to the ordinary constructs of JavaScript expressions. The notation X^*, $X^?$, and X^+ mean zero-or-more, zero-or-one, and one-or-more occurrences of X, respectively. The meta-variable s ranges over strings, i ranges over positive integers, and j and k range over integers.

required transformations. For this purpose, we introduce the notion of a *code template*, which is an incomplete JavaScript expression that has one or more missing pieces (or holes) that must be instantiated with other JavaScript expressions for the template to become a syntactically valid JavaScript expression.

We can view a code template as a form of meta-program that takes one or more expressions as input and then interpolates these expressions into the holes of the template to form a valid JavaScript expression. The key idea behind the templating mechanism is that the holes of the template are instantiated with code from the vicinity of the location in the client code that is matched by the detection pattern ρ. A detection pattern can either match a call, a property read, a property write, or a module import. In a transformation, parts of the subtree of the matched AST node have to be replaced (as in Example 1) or, in some cases, the kind of the matched AST node has to change (as in Example 2, where a property write operation is changed into a method call). In either case, the variable names and literals used in the original client code typically also have to appear in the transformed version of code for the transformation to be correct. For example, it is essential that the function written to the `drain` property on line 23 is the same function passed to the `drain` function on line 24.

To facilitate these kinds of transformations, we introduce an AST reference notation that is used to specify both the holes of the templates and how expressions should be retrieved for these holes. The idea is that one can use this notation to interpolate expressions into the template, where these expressions are retrieved relative to the AST node matched by ρ. For example, if ρ matches a call node, then an AST reference can be used to obtain, for example, the receiver or the arguments of that call. While AST references technically reference AST nodes, it is often more convenient to think of them as references to expressions since all the allowed AST references always point to nodes whose subtrees form expressions.

Figure 2 shows the grammar for code templates. A code template is a JavaScript expression (*Expression*) that can contain some special constructs explained in the following.

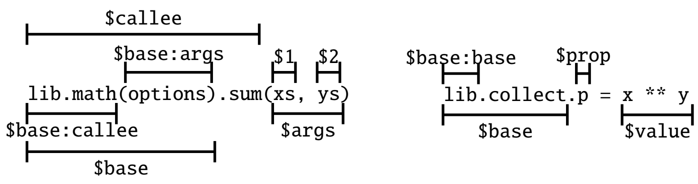

Fig. 3: AST reference examples.

AST references An AST reference consists of a '`$`' symbol followed by a list of '`:`'-separated elements specifying which node is referenced relative to the node matched by ρ. Six kinds of AST reference elements (*RefElement*) are available:

- `prop` refers to the property name p in a property access $e.p$ of a method call, property read, or property write expression.
- `value` refers to the right-hand-side expression e_2 of a property write expression $e_1.p = e_2$.
- `base` refers to the receiver value e in a property access $e.p$ of a method call, property read, or property write expression.
- `callee` refers to the function value of a function, method, or constructor call.
- i refers the i'th argument of a call.
- `args` refers to all the arguments of a call.

Example 4 In Figure 3, we show examples of which nodes various AST references refer to, relative to a call node (left) and a property write node (right). Notice how '`:`' is used to combine references: `$base:base` refers to the receiver of the receiver, `$base:callee` refers to the function that is called to compute the receiver of the `sum` method call, and `$base:args` refers to the arguments passed to this function.

Not all the different kinds of AST references make sense for every kind of detection pattern, for example, `$value` is only meaningful if transforming a property write. If a user writes a template that is invalid for a node matched by the detection pattern, for example, if `$value` is used when transforming a method call, or if `$3` is used when transforming a call with fewer than three arguments, then JSFIX is unable to perform the transformation and instead reports an error.

The syntax `$args[`j`, `k`]` denotes a slice of the arguments, from the j'th until (and including) the k'th argument. (The notation `$`j can thus be seen as an abbreviation of `$args[`j`, `j`]`.) Negative numbers count from the right, for example -1 denotes the last argument. The variant `$args[`j`,]` refers to every argument from j and onwards.

Example 5 Consider the call expression on the left of Figure 3. We can obtain a reference to the individual arguments of `sum` using the argument index reference, for example, `$2` refers to `ys`. We can similarly select slices of the arguments. For example, `$args[1, 2]` results in the slice `xs, ys`, and `$args[-2, -2]` results in the slice `xs`. Being able to select arguments counting from right to left is sometimes needed for selecting the last argument in a variadic function as we demonstrate in Example S1. (Examples whose name begin with 'S' can be found in the supplementary material [5].)

Example 6 Continuing Example 1, the calls to `max` that have to be transformed are exactly those calls where `max` is called with three arguments and the second argument is a function. Hence, the following semantic patch will automate the update of the `max` calls:[5]

```
call <lodash>.max [3, 3] 2:function ⇝ $callee($1, $2.bind($3))
```

The detection pattern matches the calls to the `max` function on the *lodash* module object, where `max` is called with exactly three arguments and the second argument has type `function`.[6] The code template specifies that those calls must be transformed such that the method call (where `$callee` refers to the function value) has the same first argument (`$1`), but where the second argument is the result of calling `bind` on the old second argument (`$2`) passing the old third argument (`$3`) as an argument to `bind`. Hence, the transformation that is applied is exactly the one shown on line 20 in Example 1.

Example 7 Continuing Example 2, the following semantic patch expresses the required transformation:

```
write <async>.queue().drain ⇝ $base.drain($value)
```

The detection pattern matches writes of the `drain` property on objects returned by calls to the `queue` function on the *async* module object. These writes are transformed such that `drain` is instead invoked on the `queue` object (referenced by `$base`), such that the value previously written to `drain` (`$value`) is passed as an argument to `drain`.

The notation $[s_1 \Rightarrow s_1', \ldots, s_n \Rightarrow s_n']$ allows us to express identifier replacements, as shown in the following example.

Example 8 Among the breaking changes in *lodash* 4, the function `any` is renamed to `some`, and `all` is renamed to `every`. We can capture both using a single, concise semantic patch (where `{any,all}` matches either `any` or `all`):

```
read <lodash>.{any,all} ⇝ $base.$prop[any ⇒ some, all ⇒ every]
```

Module imports Many breaking changes in libraries involve the structure of their modules. With the npm module system, modules are files that are loaded using `import` (as in Example 3) or using `require` (as in Example 1). We provide the notation <*ModuleElement*$^+$> for transforming and adding module loading. As an example, the code template `<rxjs/operators>.find` will generate code that ensures that the `'rxjs/operators'` module is loaded and then access its `find` property. Using this special notation instead of simply using calls to `require` in code templates to load modules has several advantages: (1) it will move all module loads to the outer-most scope, which is the more idiomatic way to load modules in JavaScript; (2) if loading a module from a package that the client currently does not depend upon, JSFIX will add that package as a dependency to the client's package.json file; (3) it will ensure that the same module is not loaded multiple times, which could happen if using `require` in a code template and that code template is used for multiple transformations in the same file; and (4) it will use the same style of module loading as the client already uses, i.e., `require` or `import`.

The detection patterns supported by TAPIR for specifying module names can contain wildcards (e.g. `*`) and sets of filenames (e.g. `{find,map}`). To allow the code templates to refer to the corresponding parts of the module names, we provide the notation $\#i$ to refer to the i'th non-constant component of the detection pattern. (This mechanism is inspired by the use of capturing groups and backreferences in traditional regular expression notation.)

Example 9 Continuing Example 3, we can express the required transformation with the following semantic patch:

```
import rxjs/operator/{find,map} ⇝ <rxjs/operators>.#1
```

The detection pattern matches imports from the modules `'rxjs/operator/find'` and `'rxjs/operator/map'` (the wildcard `{find,map}` matches both `'find'` and `'map'`), and the code template says that these imports must be transformed to reads of, respectively, the `find` or `map` property from the `'rxjs/operators'` module. Notice we use the `#1` syntax to refer to the string matched by `{find,map}` in the detection pattern.

Example 10 In some cases, adapting code to a breaking change is only possible if the client adds a new dependency. For example, in version 3.0.0 of the *uuid* library, the `parse` and `unparse` methods are removed, and clients must use the *uuid-parse* package if they want to keep using the removed methods. When using the <M> syntax, JSFIX will automatically add the dependency that contains the module M, and therefore the transformation can be expressed using this semantic patch:

```
call <uuid>.{parse,unparse} ⇝ <uuid-parse>.$prop($args)
```

Semantic patches can also be useful for post-processing transformed code to improve its readability and performance as demonstrated by Example S2 [5].

V. Interactive Phase

The TAPIR analysis that we use for the analysis phase is designed to over-approximate the AST nodes, which means that we can generally trust that no locations are missed (see the discussion about correctness assumptions at the end of this section), but some mechanism is needed for removing spurious matches. JSFIX does not require the user to manually inspect all the potential matches found in the analysis phase, to eliminate the false positives before the transformation phase. Instead, it generates a set of questions to the user about the client code, and then performs the filtering based on the answers to these questions. The questions only concern the library usage at the potentially affected locations, and the user generally does not need to be aware of the specifics of the breaking changes. In practice, JSFIX asks 0.7 questions per code transformation on average (see Section VI).

There are two main sources of precision loss in the TAPIR static analysis that cause JSFIX to ask questions to the user. The first kind of precision loss (here named **OBJ**) occurs when the static analysis is unsure if the object, on which the potentially

[5]The detection pattern of the semantic patch can easily be extended to also match many other functions affected by the `thisArg` breaking change; see breaking change number 4 for *lodash* in the supplementary material [5].

[6]The `[`j`, `k`]` call filter restricts the pattern to only match calls with between j and k arguments, i.e. calls to `max` with exactly 3 arguments in this case.

broken operation takes place, is the right type of object. To make it possible for client developers to use JSFIX without requiring them to understand the semantic patch language, we let the authors of the semantic patches manually write the questions in a client-understandable style.

Example 11 Consider the code

```
const f = (x, y) => x.map(y);
```

and the following semantic patch for *rxjs* version 6:

```
call <rxjs>?**.map ⇝ $base.pipe(<rxjs/operators>.map($args))
```

The detection pattern matches reads of the `map` property on any chain of operations on the *rxjs* module. The detection pattern also contains a '?', which means that, in particular, the expression `x.map(y)` will match even when TAPIR is too imprecise to detect whether the `x` object comes from *rxjs*. TAPIR marks such matches as *low confidence*, which causes JSFIX to ask the user for validation. If `x` happens to be, for instance, an ordinary JavaScript array rather than an object from *rxjs*, then applying the transformation would break the code. For this example, JSFIX will ask the question "*Is the receiver an rxjs observable?*" (together with the source code location of the match), and only apply the patch if the answer is "yes".

The second kind of precision loss (here named **CALL**) occurs when TAPIR is unable to determine if constraints on arguments (so-called call filters) are satisfied.

Example 12 Consider the following code:

```
29  var _ = require('lodash');
30  const f = (x, y) => _.pick(x, y);
```

The `pick` function from *lodash* is called with two arguments, an object and a property selector, and it returns an object with all the properties that satisfy the selector. The selector can either be a predicate function, or a list of strings such that all property names that appear in that list are selected. For the former case, clients should replace calls to `pick` with calls to `pickBy` when upgrading to *lodash* version 4. This transformation is captured by the following semantic patch:

```
call <lodash>.pick [2, 2] 2:function ⇝ $base.pickBy($args)
```

For this example, JSFIX will automatically generate the question "*Is the argument `y` of type function in line 30?*", and only perform the transformation if the answer is "yes". Notice how answering the question does not require any knowledge about the breaking change, but only some basic understanding of the client source code. Therefore, the client developer who is familiar with the code will likely find the question easier to answer compared to manually trying to understand the breaking change and performing the transformation. In particular, the developer would have to consider anyway whether argument `y` is of type function in line 30.

While the primary purpose of the interactive phase is to filter away spurious AST nodes to avoid redundant or wrong transformations, JSFIX also has support for two other categories of questions used when: (1) the detection patterns are too coarse-grained to determine if a breaking change can occur (see Example S3 [5]), and (2) a breaking change has relatively minor implications that some clients may prefer not to fix (see Example 13). We refer to these two categories as **EXTRA** and **MINOR**, respectively.

Example 13 The *node-fetch* library is a polyfill for the browser HTTP API `window.fetch`. In the update of *node-fetch* to version 2, the `json` method on response objects is modified such that it throws an error instead of returning an empty object when the HTTP response code is 204. A semantic patch for this breaking change can be expressed as follows.

```
call <node-fetch>?**.json
        ⇝ ($base.status === 204 ? {} : $base.json())
```

The transformed code calls `json` if the response is not equal to 204 and otherwise results in an empty object.

While this transformation is semantically correct, it is unlikely to be the desired solution for the client developer. A more idiomatic solution would be to catch the error, and then handle it accordingly. For that reason, we allow semantic patches to be marked as "low priority". For such patches, instead of always asking for each potential match, the user now also gets the options to select "yes to all" or "no to all".

Note that it is only the authors of the semantic patches who need to understand the semantic patch language – the client developers who run JSFIX only need to respond to the questions in the interactive phase about the behavior of the client code.

In the end, the correctness of the applied transformations of the client code relies on three assumptions: (1) The semantic patches correctly model all the breaking changes in the library, (2) the user answers correctly in the interactive phase, and (3) TAPIR has no false negatives (i.e., it does not miss any matches) when analyzing the client code. The evaluation presented in the following section shows that these assumptions can be satisfied in a realistic setting involving real-world libraries and clients. Furthermore, even if the transformations may not be 100% correct and require manual review for some clients, the automatically transformed client code can still serve as a good starting point compared to the traditional fully manual practice.

VI. Evaluation

We have implemented JSFIX in only 1 200 lines of TypeScript. Apart from the detection analysis and transformation phase as presented in Section IV, JSFIX also performs some auxiliary tasks to improve the transformed code. For example, imports that are unused after the transformations and multiple imports of the same module are automatically removed. We evaluate JSFIX by answering the following research questions:

RQ1 For how many of the breaking changes in major updates of widely used npm packages can the patch be expressed as a code template, and how complex are the code templates?

RQ2 For clients that are affected by breaking changes in a library, do the applied transformations make them compatible with the new version of the library? Are the transformations of sufficient quality that client developers are willing to accept them as pull requests?

RQ3 How many questions and which types of questions does JSFIX typically ask the user in the interactive phase?

The part of JSFIX concerning detection patterns is evaluated previously [1] and is therefore not considered in this evaluation.

Experiments To address RQ1, we first performed an experiment where we attempted to write semantic patches for as many as possible of the breaking changes appearing in major updates of widely used npm packages. The full list of semantic patches is shown in the supplementary material [5]. We were able to write these in only a few days without expert knowledge of any of the libraries. For RQ2 and RQ3, we performed a second experiment to test if JSFIX can, based on the semantic patches written in the first experiment, patch clients whose test suites fail when switching to the new library versions. Also for RQ2 and RQ3, we finally performed a third experiment, where we created pull requests with the transformations generated by JSFIX for a number of clients, to determine if the quality of the transformations is acceptable to client developers. The two latter experiments are a best-effort attempt to establish some confidence that the transformations created by JSFIX are correct; in the second experiment by showing that the transformations are correct enough to fix the broken test suites and not introduce new test failures, and in the third experiment by showing that the reviewers of the pull requests trust the transformations enough to accept them.

Benchmark selection Our experiments for answering the research questions are based on 12 major updates of top npm packages (selected from the TAPIR benchmark suite [1]), shown in the first column of Table I.[7]

To address RQ2 and RQ3, we selected the 89 clients from the evaluation of TAPIR that are known to be affected by one of the 12 major updates.[8] Since the test suites of the clients succeed prior to the update but fail afterwards, we know that these clients are affected by some of the breaking changes. Therefore, these clients require patches to become compatible with the new major version of the library.

For the third experiment, we used only those 41 of the 89 clients where no version of the client existed that already depended on the new major version of the library with breaking changes. (Comparing the patches that have been applied to the already updated clients with patches generated by JSFIX remains an interesting opportunity for future work.) In addition, we randomly selected 10 clients for each benchmark that, at the time of the selection, depended upon the benchmark in the major-range below the one of the benchmark, e.g., for the *lodash* 4.0.0 benchmark, we selected 10 clients that depend on some version of *lodash* below 4.0.0 but at least 3.0.0. We also required that these clients were updated within 180 days, since maintainers of recently updated libraries are presumably more likely to react to pull requests. For *express*, we could only find such 4 clients, so in total we consider 155 clients of the 12 libraries for the pull request experiment.

[7]We omitted three of the TAPIR benchmarks because they contain only a few breaking changes that are all unlikely to require any form of patching (for example, changes to formatting of a help message in the *commander* library) and are therefore not interesting for JSFIX.

[8]In the evaluation of TAPIR, 115 clients were considered, but some of them are using the omitted libraries mentioned in footnote 7, and some of them are written in languages that compile to JavaScript, which means that JSFIX cannot patch the source code of those clients.

A. RQ1 (Expressiveness of transformations)

A total of 326 detection patterns were required for describing the breaking changes in the 12 libraries. Most of these detection patterns are identical to the patterns from the TAPIR paper [1], but in some cases we had to split a pattern if, for example, different code templates were required depending on the number of function arguments at a call pattern.

A summary of the results for each library update is shown in Table I, where "**BC**" is the number of breaking changes in the update, "**Patterns**" is the number of detection patterns written, "**Temp**" is the number of code templates written, "**U**" (Unexpressible) is the number of detection patterns for which the required transformation cannot be expressed in our code template language, "**NGP**" (No general patch) is the number of detection patterns where, according to our knowledge of the breaking change, no single fix exists that applies to every location affected by that breaking change, "**?**" (Unknown) is the number of breaking changes for which the changelog and associated resources like GitHub issues were too incomplete for us to understand the breaking changes well enough to write a correct semantic patch. Representative examples of breaking changes in categories "Unexpressible" and "No general patch" are shown in Examples 14 and 15.

Example 14 As an example of a breaking change in the "Unexpressible" category, the changelog of the web framework *express* for version 4 contains this item: "*app.router - is removed*." While it is easy for JSFIX to detect where `app.router` is used, upon closer inspection it turns out that the breaking change has larger implications. The whole semantics around routing has changed such that the order in which routes (HTTP end-points) and middleware (plugins run as intermediate steps in the request/response cycle) are registered on the *express* application object needs to change. Consider the following excerpt of an application that uses *express* version 3:

```
31 var app = require('express')();
32 app.use(app.router);
33 // middleware
34 app.use(function(req, res, next) { ... });
35 ...
36 // routes
37 app.get('/' ...);
38 app.post(...);
```

In *express* version 3, the code `app.use(app.router)` makes *express* handle the registered routes before the middleware registered in later calls to `app.use` in the request/response cycle. In version 4 of *express*, the call to `app.use` on line 32 must be removed. However, due to the change in the ordering semantics, the call to `app.use` on line 34 must move below the calls to `app.get` and `app.post` on lines 37 and 38. Such a change is not currently expressible in the transformation language, and probably also out of scope for what an automated technique can realistically be expected to handle. To perform this change,

TABLE I: Experimental results for RQ1.

				No code template		
Library	**BC**	**Patterns**	**Temp**	**U**	**NGP**	**?**
lodash 4.0.0	51	123	123	0	0	0
async 3.0.0	4	7	6	1	0	0
express 4.0.0	18	24	23	1	0	0
chalk 2.0.0	3	3	3	0	0	0
bluebird 3.0.0	8	12	7	2	3	0
uuid 3.0.0	1	1	1	0	0	0
rxjs 6.0.0	26	55	53	0	2	0
core-js 3.0.0	26	41	35	0	5	1
node-fetch 2.0.0	9	9	7	0	2	0
winston 3.0.0	23	30	26	1	3	0
redux 4.0.0	2	2	1	0	1	0
mongoose 5.0.0	14	19	13	2	3	1
Total	**186**	**326**	**298**	**7**	**19**	**2**

a total ordering of how middleware and routes are registered to the *express* app is required, which is not easily obtained. Notice that JSFIX can still detect reads of `app.router`, so the user is being notified about this breaking change, but the transformation must be applied manually.

Example 15 An example of a breaking change in the "No general patch" category appears in the *node-fetch* HTTP library. In version 1 of *node-fetch*, clients could use the `getAll(name)` method on header objects to get an array of all header values for the `name` header. In version 2 of *node-fetch* this method is removed, so clients must now resort to using the `get(name)` method that instead returns a comma-separated string value of the `name` header values. It might seem that this breaking change is easily fixed by replacing `getAll` with `get` and then splitting the resulting string at all commas:

```
call <node-fetch>?**.getAll ⇝ $base.get($args).split(',')
```

However, since commas may also appear inside each of the header values, the resulting array may not be correct. Since there are no other value separators in the string, this breaking change does not have a general patch.

The number of breaking changes is smaller than the number of detection patterns, because we count each bullet in the changelogs as one breaking change (as in [1]). Some bullets may only concern a single method or property, while others concern tens of methods or properties. Hence the number of breaking changes should only be viewed as a weak indicator of how extensive the impact of each major update is.

We were able to write code templates for 298 of the 326 detection patterns. The remaining 28 patterns fall into three different categories: (1) For 7 patterns, the code template language is not expressive enough to describe the required transformation (see Example 14); (2) for 19 patterns, according to our knowledge of the breaking change, no general patch exists that will work for all clients (see Example 15); (3) for 2 patterns, the breaking change was not documented sufficiently well for us to write a correct template.

For the 298 cases where we successfully managed to write a code template, the biggest challenge was to understand how to address the breaking changes. Not all changelogs specify in detail how clients should migrate, so we sometimes had to rely on, for example, observations of how existing clients have upgraded. For example, the update of *uuid* to version 3 removes the `parse` and `unparse` methods, but does not specify what clients should use as alternatives. While searching for solutions, we came across a package named *uuid-parse*, which contains exactly the two methods removed from *uuid* in version 3. Writing the required semantic patch that replaces the `parse` and `unparse` method calls from *uuid* with calls to the methods from *uuid-parse* was then a simple matter (see Example 10). This again demonstrates one of the strengths of our approach: Instead of requiring every client developer to understand the details of the breaking changes, once a semantic patch has been written, it can be reused for many clients.

Example 16 Some of the breaking changes required more sophisticated semantic patches. Consider the `whilst(test, iteratee, callback)` function of the *async* library, which implements an asynchronous while loop where `iteratee` (the loop body) is called as long as `test` (the loop condition) is succeeding, and `callback` is called on an error or when the iteration ends. In version 2 of *async*, the `test` function is expected to be synchronous, that is, it should provide its return value through a normal return statement. However, in version 3 of *async*, `test` is expected to be asynchronous, and must therefore instead provide its return value by calling a callback. The modifications required are expressed by the following semantic patch:

```
call <async>.whilst ⇝                                         (1)
$callee( function() {                                         (2)
  const cb = arguments[arguments.length - 1];                 (3)
  const args = Array.prototype.slice.call(arguments,0,        (4)
               arguments.length-1);                           (5)
  try {                                                       (6)
    cb(null, $1.apply(this, args));                           (7)
  } catch (e) {                                               (8)
    cb(e, null);                                              (9)
  }                                                          (10)
}, $2, $3)                                                   (11)
```

The code template wraps the test function in a new function (lines 2–11), which extracts the new callback (line 3), calls the old test function, and passes the result to the callback (line 7). Using a wrapper function like this is unlikely to be the preferred choice if a developer were to perform the update manually. In that case, a simpler and more idiomatic solution would be to modify the test function itself by adding the callback to its argument list, and replacing all of its return statements with a call to this callback. However, that solution will only work if the definition of the test function is available and never throws an error, which may not be the case if, for example, the test function is imported from a library, which is why we resort to using the more general wrapper function.

It is not always possible to express a transformation that preserves the semantics (see Example 15). However, the experiments show that such situations are rare, so this does not pose a major threat to the applicability of the technique.

In conclusion, we have found that writing the templates is relatively simple for most breaking changes, but that some code templates have to be quite general and non-idiomatic to preserve the semantics in every case. While writing a template is sometimes more difficult than transforming the client code manually, the fact that the template is reusable across all clients of the library, makes the investment worthwhile.

B. RQ2 (Correctness and quality of transformations)

Client test suites experiment To test if JSFIX can repair broken clients, we ran JSFIX on 89 clients whose test suite succeeded before the update and failed when switching to the new version of the library. We used JSFIX to patch the client code, and then we checked if the test suite of the patched client passed. This is of course not a guarantee that the patches are correct. However, if none of the test suites fail due to missed or wrong transformations, it is a strong indication that JSFIX can successfully patch the client code. To increase confidence further, we also consider feedback from client developers (see the pull request experiment below).

The results of this experiment are shown in the first 11 columns of Table II: "**C**" is the number of clients, "**Tr**" is the number of transformations done by JSFIX, "✓" (resp. "✗") is the number of client test suites succeeding (resp. failing) after the patching, "−**CT**" is the number of clients that are affected by a breaking change for which it is impossible to write a correct template in our semantic patch language, and "**Time**" is the average time (in seconds) used for the detection and patching phases per client, excluding time spent on parsing the client code. The last four columns are described in Section VI-C.

The patches produced by JSFIX are successful in making 82 of the 89 clients pass their test suites. Of the remaining 7 clients, 5 are affected by breaking changes for which it is not possible to write a correct template. For example, the three *express* clients in this category are affected by the breaking change presented in Example 14. The remaining 2 clients do not fail due to unhandled breaking changes, but they contain testing code that is indirectly affected by changes to the library's API. For example, the *rxjs* client whose test suite fails asserts that some specific properties exist on `rxjs.Observable.prototype`, however, these properties have been removed and JSFIX has removed all usages of those properties, so the assertions can safely be removed. Similarly, a test in a *redux* client fails due to a bug fix in *redux* such that a message is no longer written to `console.error`. As such a bug fix is not considered a breaking change, it is not the responsibility of JSFIX to address this problem.

For the 84 clients where all code templates were expressible, JSFIX made a total of 451 changes to the client code. The number of changes differs substantially between the benchmarks, ranging from 1 per client for *node-fetch*, *redux*, and *mongoose* to 33 per client for *rxjs*. This discrepancy is expected since the number of breaking changes varies considerably between major updates as shown in Table I. The likelihood of a client using a broken API also fluctuates across the benchmarks. For example, *rxjs* has many changes in commonly used APIs, and therefore updating clients of *rxjs* is a cumbersome and time-consuming task, which may explain why developers released *rxjs-compat* and a migration guide along with the changelog. However, the *rxjs* breaking changes are also easily expressible as semantic patches, which means that JSFIX could patch all of the breaking changes in the 6 clients.

The detection and patching took on average 1.53s per client (excluding parsing time), so JSFIX is clearly efficient enough to be practically useful. Most of the time is spent by the analysis (TAPIR), whereas the patching took on average only 0.11s.

We thereby conclude that JSFIX is almost always successful in producing code transformations that cause client test suites to succeed, and that it does so using relatively little time.

Pull request experiment We also investigated the quality of the transformations by creating pull requests of the updates produced by JSFIX to see if the transformations created by JSFIX are acceptable to the client developers. We conducted this experiment by first forking the client, then running JSFIX on the forked client, and eventually, manually performing some styling fixes to satisfy the linter of the client if necessary. The styling fixes had to be done manually since the code style convention varies from client to client. We then created a pull request based on these changes.

We first created pull requests for 41 clients from the previous experiment that had not already updated the benchmark libraries. So far, 4 of these pull requests have been accepted and 2 have been rejected. The rejections were not due to specific issues within the pull requests, but because the client developer was not willing to risk breaking the application by updating the dependency. Many of those 41 clients are no longer maintained, which probably explains why the maintainers reacted to only 6 of the pull requests. We therefore extended the experiment by adding an additional 114 clients (10 for each library, except for *express* as mentioned earlier) that had been updated within 6 months of the experiment. The results for these pull requests are shown in the last 9 columns of Table II. For each library, we show the number of pull requests ("**PR**"), the number of accepted pull requests ("**Acc**"), the number of closed (i.e., rejected) pull requests ("**Rej**"), the number of transformations ("**Tr**"), and the average time (in seconds) used for detection and patching per client, excluding parsing time ("**Time**"). The remaining columns are described in Section VI-C.

Of the 114 pull requests created, 43 involved one or more transformations. For the remaining 71 pull requests, JSFIX did not find any source locations in the client code affected by breaking changes. For these clients, the pull request messages state that the code was not affected by any breaking changes, and the only file change was updating the version of the relevant dependency in the package.json file. So far, 27 of the pull requests have been accepted (in addition to the 4 mentioned above). For 16 of these 27 pull requests, JSFIX did not find any pattern matches in the client code. Only 4 have been rejected, and only 1 of these was affected by breaking changes. The maintainer of the *async* client who rejected a pull request did end up updating the code manually (at the same source locations as transformed by JSFIX), but using more idiomatic transformations. The transformations made by JSFIX were similar to the transformation shown in Example 16, so, as explained in that example, a more

TABLE II: Experimental results for RQ2 and RQ3.

	Client test suite experiment										**Pull request experiment**								
Library	**C**	**Tr**	✓	✗	**−CT**	**Time**	OBJ	CALL	EXTRA	MINOR	**PR**	**Acc**	**Rej**	**Tr**	**Time**	OBJ	CALL	EXTRA	MINOR
lodash	13	70	13	0	0	1.76	6	2	3	4	10	1	0	25	2.01	5	10	7	3
async	8	10	8	0	0	2.29	2	0	0	0	10	2	1	2	0.63	0	0	0	0
express	10	18	7	0	3	1.19	11	0	0	2	4	1	0	17	5.89	13	0	0	2
chalk	10	54	10	0	0	0.21	0	0	0	0	10	3	0	0	0.78	0	0	0	0
bluebird	9	18	9	0	0	0.28	3	0	18	3	10	3	1	2	1.21	0	0	24	1
uuid	1	2	1	0	0	0.14	0	0	0	0	10	3	0	0	0.95	0	0	0	0
rxjs	6	200	5	1	0	2.20	40	0	0	0	10	3	0	112	1.18	106	0	0	0
core-js	8	28	8	0	0	6.25	2	0	0	0	10	4	0	140	17.46	5	0	0	0
node-fetch	8	7	7	0	1	0.46	7	1	6	17	10	3	1	3	0.24	11	0	20	14
winston	8	36	7	0	1	0.80	16	14	7	7	10	2	0	19	1.12	34	50	10	4
redux	4	4	3	1	0	0.33	0	0	0	3	10	2	0	0	3.59	0	0	0	6
mongoose	4	4	4	0	0	0.29	5	0	5	1	10	0	1	5	1.81	16	6	18	3
Total	**89**	**451**	**82**	**2**	**5**	**1.53**	**92**	**17**	**39**	**37**	**114**	**27**	**4**	**325**	**2.92**	**190**	**66**	**79**	**33**

idiomatic transformation was possible. For the other rejected pull requests, the client developers did not report any issues with the pull request. For the 11 accepted pull requests that required modifications, manual styling fixes were only applied to two of them. Example S4 describes some of the accepted pull requests [5].

In total, JSFIX made 325 transformations across the 114 clients. Most of the transformations were applied to clients of *core-js* and *rxjs*. Since the experiments indicate that 43 out of 114 clients required transformations as part of the updates, we can conclude that breaking changes impact a significant proportion of clients, motivating the need for tools like JSFIX.

Overall, the average time is less than 3 seconds. Again, almost all the time is spent by the analysis phase, whereas patching takes on average only 0.06s.

In summary, based on the client test suites experiment, we conclude that transformations produced by JSFIX are generally trustworthy. Based on the pull request experiment, we can furthermore conclude that the transformations are generally of a high enough quality that developers are willing to use them in their code.

C. RQ3 (Questions asked by JSFIX*)*

The interactive phase of JSFIX is evaluated by looking at the questions asked during the two experiments described in Section VI-B. The number of questions in each of the four categories OBJ, CALL, EXTRA, and MINOR from Section V is shown in the last four columns of the "**Client test suite experiment**" and "**Pull request experiment**" sections of Table II. All the questions have been answered by the authors of JSFIX. The questions in the first three categories are not subjective, as they all concern well-defined properties of the possible program behaviors.[9] For the MINOR category, the answers are sometimes more subjective, which means that our answers may diverge from what the client maintainer would have chosen. However, since the questions in that category concern breaking changes that have only minor implications, this divergence is unlikely to affect the client behavior in a significant way. Our answers to these questions were based on a thorough investigation of the affected client code to determine the importance of the breaking change for each client.

For the 198 clients in Table II that JSFIX transformed successfully, JSFIX asked a total of 553 questions (2.8 per client). Of these questions, 365 questions (1.8 per client) are related to imprecision in the analysis, with 282 questions (1.4 per client) being related to imprecise matching of objects and 83 questions (0.4 per client) being related to imprecise reasoning of call filters.

A total of 118 questions (0.6 per client) concern the detection patterns being too coarse-grained to accurately describe the API usage that triggers the breaking change.

For the last category, breaking changes with minor implications, there are 70 questions (0.4 per client), where the majority are asked when transforming *node-fetch* clients (see Example 13). For most of these questions, it was relatively simple to determine if the transformation should be applied by considering the client code surrounding the affected location. Unlike the other question types, a client developer needs to understand the implications of a breaking change to accurately answer these questions, but since fewer than 1 question of this type is asked per client, we do not consider them a concern for the practicality of JSFIX. It is also worth noticing that without JSFIX, the client developer would instead be forced to understand the implications of every breaking change on the client code.

Only 2.8 questions are asked per client and 0.7 questions per transformation on average. In our experience, it is easy and fast to answer the questions, even without expert knowledge of the benchmarks. To conclude, JSFIX only needs to ask a modest number of questions during the interactive phase, which makes it useful and time-saving when adapting client code to breaking changes in libraries.

D. Threats to Validity

One potential concern about the validity of the experimental results is whether we, as designers of JSFIX, are at an advantage when it comes to answering the questions in the interactive

[9] Since JavaScript is dynamically typed, it is possible that the correct answer to a question like "*Is the receiver an rxjs observable?*" is "sometimes", however, we have not observed any occurrences of this situation in our experiments.

phase. As explained in Section V, the questions only concern the client code, not the internals of JSFIX or the library code. We therefore believe that the intended user of JSFIX, which is the client developer who is familiar with the client code, will find these questions easier to answer than we did.

Another threat to validity of our conclusion about the expressiveness is that the semantic patches used in the experiments were written by the creators of JSFIX. Semantic patches written for one library can benefit numerous clients, however, it is of course important that other people can use the semantic patch language. We therefore plan to conduct user studies with library developers and other potential authors of semantic patches.

VII. Related Work

The idea of automatically transforming clients to become compatible with new library versions has been considered in previous work. The term collateral evolution, describing exactly the process of patching client code based on some formalization of the required transformation, was coined by the authors of Coccinelle [2, 3], which is designed to adapt Linux drivers to breaking changes in the kernel. It has also been adapted to Java [6]. While Coccinelle and JSFIX share many traits, the large differences between C (or Java) and JavaScript make the internals of the tools quite different. Most importantly, Coccinelle relies on the fact that C and Java are statically typed, which makes it easier to connect calls with the relevant function and method definitions than in JavaScript. The same applies to the gofix tool for Go.[10]

Others have also looked at ways to automatically compute differences between library versions, and based on these differences either suggest changes for adapting clients to breaking changes in the APIs or directly transform the client code as with JSFIX [7, 8, 9, 10, 11, 12]. Most of these approaches are for Java, where each member of a class has a fully qualified name, which makes it tractable to compute differences on a type-level between two versions of a Java API, and thereby automatically identify many breaking changes. The only approach for a dynamically typed language is the PyCompat tool for Python [12]. While it is fully automatic, it is limited to a set of 11 different kinds of breaking changes, which all concern removals, moves, and additions of fields, parameters, and classes. In particular, these approaches generally do not handle behavioral changes (sometimes called semantic changes) where the behavior of a function changes without affecting its signature. The same limitation does not apply to JSFIX, which, as the evaluation shows, is able to patch almost all breaking changes appearing in library updates. The public interface of a JavaScript library can change dynamically and has no access modifiers, which makes it difficult to statically compute changes in the interface, and therefore the existing work for other languages will not directly work for JavaScript.

The idea of using a templating mechanism for specifying program transformations has been explored in other settings. With the Spoon framework [13], Java program transformations can be specified in Java code that directly manipulates the AST, or as class templates, which are classes with holes that must be instantiated with program elements to form valid Java classes. The class templates of Spoon resemble the code templates of semantic patches. Because Java is statically typed, the holes in the class templates also have a type, and Spoon can check that the program elements used in the substitution adhere to types of the holes. The lack of static type checking in JavaScript makes that difficult in our setting, however, it may be interesting in future work to look at opportunities for validating semantic patches, for example by attempting to check that any transformation resulting from a match of a semantic patch is syntactically valid JavaScript code.

JSFIX can be viewed as a specialized program repair tool, although it is designed to prevent rather than fix bugs. Unlike, for example, template based program repair techniques [14], the patches produced by JSFIX are almost always successful, as demonstrated by the experimental evaluation, and it does not require test suites.

The concept of breaking changes has been explored extensively in previous work. Several papers have shown that breaking changes are common in both patch and minor updates that are supposed to be backward compatible [15, 16, 17, 18, 19]. A few tools have also been designed for automatically detecting breaking changes in library updates [15, 20, 21]. The JSFIX approach is designed on the premise that the semantic pattern designer is already aware of where and how breaking changes appear. However, by combining our approach with existing tools, such as NoRegrets [21], it might be possible to derive the detection pattern part of the semantic patches automatically, which may reduce the overhead of writing semantic patches.

VIII. Conclusion

We have presented an approach to automate much of the work involved in adapting JavaScript programs to evolving libraries. It is based on a notion of semantic patches that combines the pattern matching technique from TAPIR [1] with a specialized notation for code templates, thereby making it possible to express how to find and patch locations in the client code that are affected by breaking changes in the libraries.

An extensive experimental evaluation on real-world libraries and clients using our implementation JSFIX demonstrates that the code template language is sufficiently expressive to precisely capture most breaking changes, and that most semantic patches are relatively simple. As an alternative to the current practice, the manual effort required by the client developers is reduced to answering a few questions about the program behavior. On average, only 2.7 questions are asked while patching 3.8 locations per client. The tool is fast and produces useful patches for the broken clients. In particular, 31 pull requests (many involving substantial changes to the client code) produced directly from the output of JSFIX have already been accepted by client developers.

Acknowledgments This work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 647544).

[10]https://golang.org/cmd/fix/

References

[1] A. Møller, B. B. Nielsen, and M. T. Torp, "Detecting locations in JavaScript programs affected by breaking library changes," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 187:1–187:25, 2020.

[2] Y. Padioleau, R. R. Hansen, J. L. Lawall, and G. Muller, "Semantic patches for documenting and automating collateral evolutions in Linux device drivers," in *Proceedings of the 3rd Workshop on Programming Languages and Operating Systems: Linguistic Support for Modern Operating Systems, PLOS 2006, San Jose, California, USA, October 22, 2006*. ACM, 2006, p. 10.

[3] Y. Padioleau, J. L. Lawall, and G. Muller, "SmPL: A domain-specific language for specifying collateral evolutions in linux device drivers," *Electron. Notes Theor. Comput. Sci.*, vol. 166, pp. 47–62, 2007.

[4] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the JavaScript package ecosystem," in *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*. ACM, 2016, pp. 351–361.

[5] B. B. Nielsen, M. T. Torp, and A. Møller, "Semantic patches for adaptation of JavaScript programs to evolving libraries (supplementary material)," 2021, Aarhus University. [Online]. Available: https://brics.dk/jsfix/

[6] H. J. Kang, F. Thung, J. Lawall, G. Muller, L. Jiang, and D. Lo, "Semantic patches for Java program transformation (experience report)," in *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom.*, ser. LIPIcs, vol. 134. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019, pp. 22:1–22:27.

[7] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 4, pp. 19:1–19:35, 2011.

[8] H. A. Nguyen, T. T. Nguyen, G. W. Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to API usage adaptation," in *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*. ACM, 2010, pp. 302–321.

[9] L. Li, T. F. Bissyandé, H. Wang, and J. Klein, "CiD: automating the detection of API-related compatibility issues in Android apps," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*. ACM, 2018, pp. 153–163.

[10] M. Fazzini, Q. Xin, and A. Orso, "Automated API-usage update for Android apps," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*. ACM, 2019, pp. 204–215.

[11] M. Lamothe, W. Shang, and T. Chen, "A4: automatically assisting android API migrations using code examples," *CoRR*, vol. abs/1812.04894, 2018.

[12] Z. Zhang, H. Zhu, M. Wen, Y. Tao, Y. Liu, and Y. Xiong, "How do Python framework APIs evolve? an exploratory study," in *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER London, Ontario, February 18-21, 2020*. IEEE, 2020.

[13] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "SPOON: a library for implementing analyses and transformations of Java source code," *Softw., Pract. Exper.*, vol. 46, no. 9, pp. 1155–1179, 2016.

[14] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*. ACM, 2019, pp. 31–42.

[15] G. Mezzetti, A. Møller, and M. T. Torp, "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*, ser. LIPIcs, vol. 109, 2018, pp. 7:1–7:24.

[16] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. Backes, "Keep me updated: An empirical study of third-party library updatability on Android," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. ACM, 2017, pp. 2187–2200.

[17] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic versioning and impact of breaking changes in the Maven repository," *J. Syst. Softw.*, vol. 129, pp. 140–158, 2017.

[18] A. Brito, L. Xavier, A. C. Hora, and M. T. Valente, "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, 2018, pp. 255–265.

[19] K. Jezek, J. Dietrich, and P. Brada, "How Java APIs break - an empirical study," *Inf. Softw. Technol.*, vol. 65, pp. 129–146, 2015.

[20] A. Brito, L. Xavier, A. C. Hora, and M. T. Valente, "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, 2018, pp. 507–511.

[21] A. Møller and M. T. Torp, "Model-based testing of breaking changes in Node.js libraries," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. ACM, 2019, pp. 409–419.