Statically Identifying Class Dependencies in Legacy JavaScript Systems: First Results

Leonardo Humberto Silva
Federal Institute of Northern
Minas Gerais, Brazil

Marco Tulio Valente
Federal University of
Minas Gerais, Brazil

Alexandre Bergel
Pleiad Lab, DCC
University of Chile

***Abstract*—Identifying dependencies between classes is an essential activity when maintaining and evolving software applications. It is also known that JavaScript developers often use classes to structure their projects. This happens even in legacy code, i.e., code implemented in JavaScript versions that do not provide syntactical support to classes. However, identifying associations and other dependencies between classes remain a challenge due to the lack of static type annotations. This paper investigates the use of type inference to identify relations between classes in legacy JavaScript code. To this purpose, we rely on Flow, a state-of-the-art type checker and inferencer tool for JavaScript. We perform a study using code with and without annotating the class import statements in two modular applications. The results show that precision is 100% in both systems, and that the annotated version improves the recall, ranging from 37% to 51% for dependencies in general and from 54% to 85% for associations. Therefore, we hypothesize that these tools should also depend on dynamic analysis to cover all possible dependencies in JavaScript code.**

***Index Terms*—JavaScript; Reverse engineering; Class dependencies.**

I. Introduction

Accurately identifying dependencies between software components is essential in reengineering, reverse engineering, and software maintenance activities [1], [2]. In a statically typed language (*e.g.,* Java), dependencies are directly expressed from the type of a program structure (*e.g.,* method definitions, variables, class references). Dependencies are therefore trivially deduced. However, dynamically-typed languages, including JavaScript, do not offer type information which significantly raises the difficulty to extract dependencies. These dependencies form the basis to provide, for example, class diagrams for JavaScript applications.

Type inference is a known technique that identifies the type of variables using static code analysis. Several type inferencer tools have been proposed for JavaScript [3], [4]. Oddly, no attempt has been made to evaluate the accuracy of such tools to retrieve dependencies in *legacy* JavaScript programs, as far as we know. In this paper, we employ the term *legacy* to refer to applications that do not use the latest JavaScript standard, known as ECMAScript 6 (ES6), which offers syntactic support of classes. However, even without native support, classes are often emulated in legacy JavaScript using the prototype mechanism of the language. For example, in a previous work, we found that structures emulating classes are present in 74% of the studied systems [5]. It is also important to mention that most of the JavaScript projects are currently implemented according to ECMAScript 5 (ES5), the version prior to ES6, representing a large codebase of legacy code that needs maintenance and evolution.

This paper evaluates the use of Flow type inferencer to extract dependencies between classes in legacy JavaScript code. We perform a study using code with and without annotating the class import statements in two modular applications. The results show that precision is 100% in both systems, and that the annotated version improves recall, ranging from 37% to 51% for dependencies in general and from 54% to 85% for associations.

The remainder of this paper is organized as follows. Section II provides a background on class emulation in legacy JavaScript code, class dependencies, and type inference. Section III describes the methodology used to detect class-to-class dependencies in legacy JavaScript code. Section IV describes the research questions that guide this work, along with the dataset and metrics used in our study. We discuss answers to the proposed research questions in Section V. Threats to validity are exposed in Section VI and related work is presented in Section VII. We conclude by summarizing our findings and discussing future work in Section VIII.

II. Background

A. Class Emulation in Legacy JavaScript Code

Using functions is the common strategy to emulate classes in legacy JavaScript [5], [6]. Particularly, any function can be used as a template for the creation of objects. When a function is used as a class constructor, the `this` variable is bound to the new object under construction. Variables linked to `this` are used to define properties that emulate attributes and methods. If a property is an inner function, it represents a *method*; otherwise, it is an *attribute*. The operator `new` is used to instantiate class objects.

To illustrate the emulation of classes in legacy JavaScript code, we use a simple `Queue` class. Listing 1 presents the code that defines this class, which includes a constructor function (lines 2-4), one attribute `_elements` (line 3), and methods `isEmpty`, `push`, and `pop` (lines 5-7).

Indeed, the implementation in Listing 1 represents one possibility of class emulation in JavaScript. Variations are possible, like implementing methods inside/outside class constructors and using anonymous/non-anonymous functions [5], [7].

```
// Class Queue
function Queue() { // Constructor function
  this._elements = new LinkedList();
}
Queue.prototype.isEmpty = function() {...}
Queue.prototype.push = function(e) {...}
Queue.prototype.pop = function() {...}
```

Listing 1: Class emulation in legacy JavaScript code

B. Class Dependencies

Based on the UML specification [8], we consider two types of class-to-class dependencies in this work. *Associations* are particular cases of dependencies in which a class contains one or more attributes that are bound to instances of other classes. Figure 1 shows two examples of associations commonly found in JavaScript systems. In code (1), the constructor function for class `Z` is implemented in lines 4-6. The attribute `x`, that belongs to class `Z`, receives an instance of class `X` (line 5), creating an association between the two classes. In code (2), the constructor function `Z` has a parameter `x`. In line 6, an instance of `Z` is created, with `x` bound to an object of type `X`.

```
1 // fileZ.js
2 var X = require("fileX.js");
3
4 function Z() {
5   this.x = new X();
6 }
```

(1)

```
1 // fileZ.js
2 function Z(x) {
3   this.x = x;
4 }
5 // testCase.js
6 var z = new Z(new X());
```

(2)

Fig. 1: Examples of associations (from class `Z` to class `X`)

When the relationship between classes does not involve assignments of objects to class attributes, we have a *uses* relationship, because one class just uses the other. Figure 2 shows two examples of dependencies in which a class `Z` *uses* another class `X`. In code (1), we can see a method `foo` (lines 1-6) of a class `Z`. This method receives an object as argument and *uses* it to invoke its method `bar` (line 4). In code (2), the method `foo` creates an instance of `X` and stores it in a temporary variable `_x` for later *use* (line 4).

```
1 Z.prototype.foo =
2   function(x)
3   {
4     var _bar = x.bar();
5     ...
6   }
```

(1)

```
1 Z.prototype.foo =
2   function()
3   {
4     var _x = new X();
5     ...
6   }
```

(2)

Fig. 2: Examples of dependencies of type "uses"

In the remaining of this paper, the term dependency is used generically to reference associations and dependencies of the type *uses*.

C. Type Inference

A type is a collection of program entities that share common properties. In general, we refer to the process of reasoning about unknown types as *type inference* [9]. A type inference mechanism analyzes a program to infer the types of some or all of its expressions. Commonly, a type checker verifies if all the types are properly defined and used according to the semantics of the programming language. Statically typed languages, such as Java and C++, perform type checking at compile-time, demanding the developers to explicitly declare the types in the source code. Dynamically typed programming languages, such as JavaScript and Smalltalk, only check types at runtime.

Flow[1] is a static type checker for JavaScript designed by Facebook. It employs a control-flow analysis that compilers typically perform to extract semantic information from code, and then uses this information for type inference. Listing 2 is one example of code in which Flow can detect incompatible types involving an expression (line 3) and the type passed as argument of the function call (line 5). Listing 3 shows the result of applying Flow to the code in Listing 2. As we can see, Flow indicates a type error in line 3 (a string is used as an operand in a multiplication).

```
1 // Function foo expects a number as argument
2 function foo(x) {
3   return x * 10;
4 }
5 foo('Hello, Flow!');
```

Listing 2: Example of incompatible types detected by Flow

```
foo.js:5
  5: foo('Hello, Flow!');
     ^^^^^^^^^^^^^^^^^^^ function call
  3:   return x * 10;
              ^ string. This type is incompatible with
  3:   return x * 10;
              ^^^^^^ number
```

Listing 3: Warning messages from Flow for the code in Listing 2

Developers can also use type annotations to document their systems and to help Flow during type checking, although this is not mandatory.

III. Evaluated Approach

Our central goal in this paper is to describe the first results of a study that uses a static type checker (Flow) to identify dependencies between structures that emulate classes in legacy JavaScript code. Figure 3 presents an overview of the approach investigated in our study. Given a JavaScript legacy application, we perform the following steps.

Step 1: Identify classes. In the first step, we identify the classes emulated in a legacy JavaScript codebase using JSClassFinder [10], which is a tool designed to detect classes in legacy JavaScript code. JSClassFinder works on the application's abstract syntax tree that is generated after pre-processing the source code. The tool then applies a set of heuristics to identify classes, methods, and attributes [5].

Step 2: Infer types. We execute Flow passing the application's source code and tests as input. The generated output is a text file that contains the coordinates (line, column) for every element

[1]http://flowtype.org/

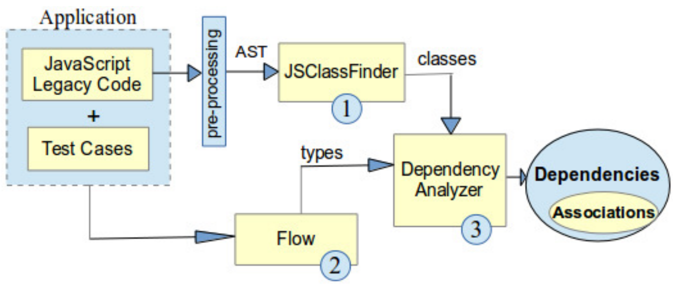

Fig. 3: Overview of the evaluated approach

of the source code, and their respective types. Listing 4 shows Flow's output for the code in Listing 2. In line 1, for example, we can see that there is a function, denoted by the arrow ("=>"), whose name is located in line 1 of the file `foo.js`, between columns 10 and 12. This function has a `string x` as argument (line 1, column 14) and returns a `number`.

```
1 foo.js:1:10-12: (x: string) => number
2 foo.js:1:14: string
3 foo.js:2:9: string
4 foo.js:2:9-14: number
5 foo.js:2:13-14: number
6 foo.js:4:1-3: (x: string) => number
7 foo.js:4:1-19: number
8 foo.js:4:5-18: string
```

Listing 4: Example of Flow's type inference algorithm output file

To provide the necessary input for Flow (step 2), the application's test cases are used together with the source code files. In this case, the tests are important to determine the types involved in class instantiations and method calls. On the other hand, the class identification (step 1) is based only on the source files, without the tests.

Step 3: Locate the dependencies. The classes (step 1) and the inferred types (step 2) are used by the component Dependency Analyzer (see Figure 3) to identify associations and *uses* dependencies. In this step, we look for inferred types that correspond to classes detected by JSClassFinder. To classify the associations we identify the types linked to class attributes.

IV. Evaluation Design

In this section, we describe a first study that aims to answer the following research questions:

- RQ1: *What is the accuracy of the proposed approach in detecting class dependencies?* Answering this research question is important to assess how accurate and complete are the class dependencies and associations identified by the approach described in Section III. We measure precision and recall to answer this first research question.
- RQ2: *Do module annotations improve the accuracy of the proposed approach?* We use this research question to analyze the impact of including type annotations in the correspondent import/export statements of the JavaScript source files. For example, in Figure 1, code (1), we can see an import statement (line 2), represented by a call to function `require()`, that assigns to variable `X` the definitions exported by `fileX.js`. Listing 5 shows the same import statement of Figure 1, code (1), but with a class type annotation. This annotation informs Flow that the variable `X` (line 1) has indeed the type `X`. We compare the measures obtained in RQ1 with the ones obtained after manually including similar annotations in all import (`require`) statements.

```
1 var X: Class<X> = require("fileX.js");
2 ...
```

Listing 5: Example of import statement with annotation

Dataset and the oracle. For our study, we need systems that emulate classes in legacy JavaScript in order to find the dependencies between these classes. We also need an oracle of class dependencies. We use two TypeScript open-source projects to build this oracle and minimize possible bias. TypeScript is an extension of JavaScript that offers a module system, classes, interfaces and a gradual type system [11]. In order to execute, TypeScript code is *transpiled* to vanilla JavaScript.[2] By parsing and extracting explicit types in TypeScript programs, class-to-class dependencies can be found and added to the oracle. The strategy of using TypeScript projects to build an oracle is also adopted by Rostami et al. [6] in their work to detect constructor functions in JavaScript.

Table I presents the main characteristics of the two selected TypeScript projects, including version number, size (LOC), number of classes, number of dependencies, and number of dependencies that are class associations. INVERSIFYJS[3] is a lightweight inversion of control container for TypeScript and JavaScript applications. SATELLIZER[4] is an end-to-end token-based authentication module for AngularJS.

TABLE I: CHARACTERISTICS OF THE ANALYZED SYSTEMS.

System	Version	LOC	# Classes	# Class Dependencies	# Class Associations
INVERSIFYJS	2.0.1	1,527	20	160	26
SATELLIZER	0.15.5	990	11	39	20

As an example of class in TypeScript, Listing 6 shows part of the implementation of class `QueryString` in INVERSIFYJS. We can see type annotations in attributes (line 2), which are used to infer associations, and in parameters (lines 4 and 7), which are used to infer *uses* relations.

```
1  class QueryableString {
2    private str: string;
3
4    constructor(str:string) {
5      this.str = str;
6    }
7    public startsWith (searchString: string): boolean {
8      ...
9    }
10 }
```

Listing 6: Example of class in TypeScript

[2]A transpiler is a source-to-source compiler. Transpilers are used, for example, to convert from TypeScript to JavaScript, in order to guarantee compatibility with existing browsers and runtime tools. The automatically generated transpiled code contains all function constructors and methods, along with their dependencies, very similar to a naturally written code.

[3]https://github.com/inversify/InversifyJS

[4]https://github.com/sahat/satellizer

V. Results

A. What is the accuracy of the proposed approach in detecting class dependencies?

As described in Section IV, we evaluate our approach using precision and recall. Table II summarizes the results according to the proposed approach. As we can see, precision is 100% in all evaluated scenarios and systems. Maximal precision values suggest that the type inference mechanism is very conservative. No false positives were found in both systems. In the case of recall, we can observe a very low result for class dependencies in INVERSIFYJS (6%) and a slightly better result if we only consider class associations (19%). Listing 7 shows one example of dependency that could not be identified in INVERSIFYJS. This example includes the implementation of method `getAllTagged` in class `Kernel`. Flow was not able to identify that the object created in line 5 represents an instance of class `Metadata`, therefore missing a dependency from `Kernel` to this class.

TABLE II: Precision and Recall results

System	All Dependencies					Associations				
	TP	FP	FN	Prec.	Recall	TP	FP	FN	Prec.	Recall
INVERSIFYJS	9	0	151	100%	6%	5	0	21	100%	19%
SATELLIZER	17	0	22	100%	44%	17	0	3	100%	85%

```
1 var metadata_1 = require("metadata.js");
2 ...
3 Kernel.prototype.getAllTagged =
4   function (serviceIdentifier, key, value) {
5         var metadata = new metadata_1(key, value);
6         ...
7   };
```

Listing 7: Example of dependency not detected in INVERSIFYJS

The results are more accurate for system SATELLIZER, where recall ranges from 44% for all dependencies to 85% when considering only associations. The better results achieved in SATELLIZER may be explained due to the fact that associations in this system represent 51% of the cases, against 16% in INVERSIFYJS.

Summary: The proposed approach has a precision of 100% (in both systems). Recall ranges from 6% to 44% for all dependencies and from 19% to 85% for associations. Further investigation is needed to understand the causes of false negatives, specially for dependencies that are not associations.

B. Do module annotations improve the accuracy of the proposed approach?

Table III summarizes the results with the import statements annotated. We have high precision values (100%), as in RQ #1. In the case of recall, we observe a significant improvement in INVERSIFYJS with 37% for all class dependencies and 54% if we only consider class associations. By contrast, system SATELLIZER is less sensitive to the presence of annotations having 51% for dependencies and preserving the same 85% for class associations.

TABLE III: Precision and recall with explicit module annotations

System	All Dependencies					Associations				
	TP	FP	FN	Prec.	Recall	TP	FP	FN	Prec.	Recall
INVERSIFYJS	59	0	101	100%	37%	14	0	12	100%	54%
SATELLIZER	20	0	19	100%	51%	17	0	3	100%	85%

Summary: The use of explicit module annotations in import statements improved the values of recall, which now ranges from 37% to 51% for all dependencies and from 54% to 85% for associations.

VI. Threats to Validity

External Validity. We studied two open-source JavaScript/TypeScript systems. For this reason, our preliminary results might not represent all possible cases of class dependencies and associations. If other systems were considered, the effect of adding module annotations could vary. To allow the replication of our study, the oracle along with the detected dependencies for both systems is available on-line.[5]

Internal Validity. In the evaluation we only address module dependencies that comply with CommonJS[6]. There are different strategies to incorporate modules into JavaScript programs, e.g., global functions and AMD[7], which also support modules that can be annotated according to our approach. Therefore, we can extend our study by including applications using other module systems.

Construct Validity. The classes emulated in the legacy code were detected by JSClassFinder [5], [10]. Therefore, it is possible that JSClassFinder wrongly identifies some structures as classes (false positives) or that it misses some classes in the legacy code (false negatives). Likewise, we rely on Flow to infer the types needed for identifying class relations. However, the results we achieved for precision and recall indicate that both tools presented consistent behavior.

VII. Related Work

In a previous work, we present a set of heuristics followed by an empirical study to investigate the prevalence of class-based structures in legacy JavaScript code [5]. The study was conducted on 50 popular JavaScript systems, implemented according to ECMAScript 5. The results indicated that class-based constructs are present in 74% of the studied systems. We also implemented a tool, JSClassFinder [10], to detect classes in legacy JavaScript code. We used this tool to identify the emulated classes in the presented paper.

[5]https://github.com/leonardo-silva/JSClassDependencies
[6]http://requirejs.org/docs/commonjs.html
[7]https://github.com/amdjs/amdjs-api/wiki/AMD

Rostami et al. [6] propose an alternative tool, called JS-Deodorant, to detect constructor functions in legacy JavaScript systems. They first identify all object instantiations, even when there is no explicit object instantiation statement (*e.g.,* the keyword `new`), and then link each instance to its constructor function. Finally, the identified constructors represent the emulated classes and the functions that belong to these constructors (inner functions) represent the methods.

Cloutier et al. [12] present a reverse engineering tool, called WAVI, that uses static analysis and a filter-based mechanism to retrieve and document the structure of a Web application. WAVI provides customized class diagrams for JavaScript, as proposed by web application extensions (WAE) [13]. However, the tool is not able to identify class constructors nor their dependencies in legacy JavaScript code.

Jensen et al. [14], [15] introduce TAJS, which is a dataflow analyzer for JavaScript that relies on allocation site abstraction for objects and constant propagation for primitive values. The authors also provide an Eclipse plug-in that can be used to catch type-related errors. Therefore, future work can extend our study using TAJS as type inferencer.

There are also tools and techniques that rely on execution traces to understand event-based interactions in JavaScript. Alimadadi et al. [16] presented a tool, called Clematis, for supporting comprehension of web applications. The tool captures a detailed trace of a web application's behaviour during a particular user session and creates a behavioural model through a combination of automated JavaScript code instrumentation and transformation. This model is then presented to the developers as an interactive visualization that depicts the creation and flow of triggered events. Zaidman et al. [17] presented a tool, called FireDetective, to record execution traces of the JavaScript code that is executed in the browser (client) and also in the server. The level of detail used is the call level: the tool records the names of all functions and methods that were called, and in what order they were called, allowing the reconstruction of a call tree representation of each trace. Although event-based interactions do not involve class relations directly, future work can use similar techniques to extend our approach by looking for dependencies in execution traces.

VIII. Conclusion and Future Work

In this paper, we propose an approach that uses type inference to identify class-to-class dependencies and associations in legacy JavaScript code. We report a study on two open-source projects to identify relations between classes and modules. Our results show that precision reaches 100% in the two systems evaluated in the paper. However, recall is lower, ranging from 6% to 44% for dependencies in general and from 19% to 85% for associations. We also show that, after manually annotating import statements with type information, Flow's recall increases, reaching 37% to 51% for dependencies and from 54% to 85% for associations, which is probably not sufficient to provide reliable reverse engineering tools to JavaScript developers. Therefore, we hypothesize that these tools should also depend on dynamic analysis to cover all possible dependencies in JavaScript code.

As future work, we intend to enrich our research in two directions. First, we plan to extend our study analyzing a larger set of JavaScript systems. In this way, we can identify other instances of missing dependencies (false negatives) and visualize other opportunities to introduce annotations. Second, we plan to improve recall by combining our approach with dynamic analysis. We can instrument JavaScript code to record execution traces using, for example, tools like Aran [18].

Acknowledgments

This research is supported by CNPq and FAPEMIG.

References

[1] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Using dependency models to manage complex software architecture," in *20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005, pp. 167–176.

[2] J. Laval, S. Denier, S. Ducasse, and A. Bergel, "Identifying cycle causes with enriched dependency structural matrix," in *16th Working Conference on Reverse Engineering (WCRE)*, 2009, pp. 113–122.

[3] B. Hackett and S. Guo, "Fast and precise hybrid type inference for JavaScript," in *33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012, pp. 239–250.

[4] C. Anderson, P. Giannini, and S. Drossopoulou, "Towards type inference for JavaScript," in *19th European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, 2005, pp. 428–452.

[5] L. H. Silva, M. Ramos, M. T. Valente, A. Bergel, and N. Anquetil, "Does JavaScript software embrace classes?" in *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 73–82.

[6] S. Rostami, L. Eshkevari, D. Mazinanian, and N. Tsantalis, "Detecting function constructors in JavaScript," in *32nd IEEE International Conference on Software Maintenance and Evolution (ICSME ERA)*, 2016.

[7] W. Gama, M. Alalfi, J. Cordy, and T. Dean, "Normalizing object-oriented class styles in JavaScript," in *14th IEEE International Symposium on Web Systems Evolution (WSE)*, 2012, pp. 79–83.

[8] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3rd ed. Addison-Wesley, 2003.

[9] J. Palsberg and M. I. Schwartzbach, "Object-oriented type inference," in *6th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1991, pp. 146–161.

[10] L. H. Silva, D. Hovadick, M. T. Valente, A. Bergel, N. Anquetil, and A. Etien, "JSClassFinder: A tool to detect class-like structures in JavaScript," in *6th Brazilian Conference on Software: Theory and Practice (CBSoft), Tools Demonstration Track*, 2015, pp. 113–120.

[11] C. Nance, *TypeScript Essentials*. Packt Publishing, 2014.

[12] J. Cloutier, S. Kpodjedo, and G. El-Boussaidi, "WAVI: A reverse engineering tool for web applications," in *24th IEEE International Conference on Program Comprehension (ICPC)*, 2016, pp. 1–3.

[13] J. Conallen, *Building Web Applications with UML*, 2nd ed. Addison-Wesley, 2002.

[14] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for JavaScript," in *16th International Static Analysis Symposium (SAS)*. Springer-Verlag, 2009.

[15] S. H. Jensen, A. Møller, and P. Thiemann, "Interprocedural analysis with lazy propagation," in *17th International Static Analysis Symposium (SAS)*. Springer-Verlag, 2010.

[16] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman, "Understanding JavaScript event-based interactions," in *International Conference on Software Engineering (ICSE)*, 2014, pp. 367–377.

[17] A. Zaidman, N. Matthijssen, M. D. Storey, and A. van Deursen, "Understanding Ajax applications by connecting client and server-side execution traces," *Empirical Software Engineering*, vol. 18, no. 2, pp. 181–218, 2013.

[18] L. Christophe, C. De Roover, and W. De Meuter, "Poster: Dynamic analysis using JavaScript proxies," in *37th International Conference on Software Engineering (ICSE)*, 2015, pp. 813–814.