Generation of TypeScript Declaration Files from JavaScript Code

John Q. Public
Dummy University Computing Laboratory, Country
My second affiliation, Country
`http://www.myhomepage.edu`
johnqpublic@dummyuni.org

Abstract

Developers are starting to write large and complex applications in TypeScript, a typed dialect of JavaScript. TypeScript applications integrate JavaScript libraries via typed descriptions of their APIs called declaration files. DefinitelyTyped is the standard public repository for these files. The repository is maintained manually by volunteers, which is error prone and time consuming. Discrepancies between a declaration file and the JavaScript implementation lead to incorrect feedback from the TypeScript IDE and thus to incorrect uses of the underlying JavaScript library.

This work presents `dts-generate`, a tool that generates TypeScript declaration files for JavaScript libraries uploaded to the NPM registry. It extracts code examples from the documentation written by the developer, executes the library driven by the examples, gathers run-time information, and generates a declaration file based on this information. To evaluate the tool, 244 declaration files were generated and compared with the declaration file provided on DefinitelyTyped. 33 files out of 244 had no differences.

2012 ACM Subject Classification Software and its engineering → Software notations and tools; Software and its engineering → General programming languages

Keywords and phrases JavaScript, TypeScript, Dynamic Analysis, Declaration Files

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Funding *John Q. Public*: (Optional) author-specific funding acknowledgements

1 Introduction

JavaScript has become the most popular language for writing web applications [6]. It is also gaining popularity for back-end applications running in NodeJS, a JavaScript-based server-side platform. JavaScript is appealing to developers because its forgiving dynamic typing enables them to create simple pieces of code very quickly and proceed on a trial-and-error basis.

JavaScript was never intended to be more than a scripting language. Hence, it lacks features for maintaining and evolving large codebases. Presently, however, JavaScript is being used for creating large and complex applications. Mistakes such as mistyped property names and misunderstood or unexpected type coercions cause developers to spend a significant amount of time in debugging. There is ample evidence for such mishaps. For example, a JavaScript code blog[1] collects experiences from developers facing unexpected situations while programming in JavaScript. Listing 1 exposes some of these unintuitive JavaScript behaviors.

The cognitive load produced by such dynamic typing (which includes unchecked property names) is not present in other languages that use build tools based on type information. This insight motivated the creation of TypeScript, a superset of JavaScript with type annotations [10]. It has become a widely used alternative among JavaScript developers, because it

[1] `https://wtfjs.com`

© John Q. Public;
licensed under Creative Commons License CC-BY

Listing 1 Unexpected JavaScript behavior - falsy values, `typeof`, `null` and `undefined` operators and type coercion

```
"0" == false; // true
true == "1.00"; // true
false == "     \n\r\t      "; // true
false == []; // true
0 == []; // true
null == undefined; // true
[1] + 1; //'11'
[2] == "2"; // true
null + undefined + [1, 2, 3] // 'NaN1,2,3'
"hello world".lenth + 1 // NaN | note 'lenth' instead of 'length'
[1, 15, 20, 100].sort() // [ 1, 100, 15, 20 ]
typeof null; // object
null instanceof Object; // false
```

incorporates features that are helpful for developing and maintaining large applications [5]. TypeScript enables the early detection of several kinds of run-time errors and the integration of code intelligence tools like auto-completion in an IDE.

Existing JavaScript libraries can be used in a TypeScript project by adding a declaration file that contains a description of the library's API in terms of types. Lots of declaration files for popular JavaScript libraries have been created in a community effort in the DefinitelyTyped repository [1]. At the time of writing this repository contains declaration files for more than 6000 JavaScript libraries. Unfortunately, most declaration files in this libraries have been manually created and maintained, which is error prone and time consuming. As TypeScript takes a declaration file at face value, discrepancies are not detected at compile time and the inaccurate code-intelligence features are misleading the programmer. Moreover, TypeScript does not perform any run-time checking on types, either, so that a discrepancy between the declaration file and its corresponding JavaScript library can lead to unexpected behavior and crashes. Such behavior can lead to developer frustration, longish debugging sessions, and decreasing confidence in the tool chain.

Some previous work tackled the problem of improving the quality of declaration files. Feldthaus et al [4] search automatically for mismatches between a declaration file and implementation code. TSTest [8] adapted feedback-directed random testing to detect discrepancies between a declaration file and a JavaScript library. Tools like TSInfer and TSEvolve [7] are designed for assisting the creation of new declaration files and supporting the evolution the declaration file when the corresponding JavaScript library gets modified, respectively. They rely on an existing static analyzer for JavaScript [**?**]. TypeScript itself developed `dts-gen`, a tool that generates a declaration file that is meant to be used as a *starting point for writing a high-quality declaration file* [2].

In this work we present the tool `dts-generate` as a first step to explore the possibilities for generating useful declaration files without the heavy lifting of static analysis. `dts-generate` comes with a framework that supports the generation of declaration files for an existing JavaScript library published to the NPM registry. The tool gathers data flow and type information at run-time to generate a declaration file based on that information.

The novelty of our tool is twofold:

1. We do not rely on static analysis, which is hard to implement soundly and precisely and which is prone to maintenance problems when keeping up with JavaScript's yearly language updates.
2. Instead we extract example code from the programmer's library documentation and rely on dynamic analysis to extract typed usage patterns for the library from the example

runs.

The contributions of this paper are as follows.

- A framework that extracts code examples from the documentation of an NPM package and collects run-time type information from running these examples.
- The tool `dts-generate`, a command line application that generates a valid TypeScript declaration file for a specific NPM package using run-time information.
- A comparator for TypeScript declaration files. This tool is necessary for evaluating our framework and also useful to detect incompatibilities when evolving JavaScript modules.
- An evaluation of our framework. We examined all 6029 entries in the DefinitelyTyped repository and found 244 sufficiently well-documented NPM packages, on which we ran `dts-generate` and compared the outcome with the respective declaration file from the DefinitelyTyped repository.

2 Motivating Example

The NPM package `route-parser` is a simple route parsing, matching, and reversing library for Javascript[2]. It has about 35,000 weekly downloads and 221 NPM packages depend on it. If a developer creates or extends TypeScript code that depends on the `route-parser` library, the TypeScript compiler and IDE requires a declaration file for that library to perform static checking and code completion, respectively. We use `dts-generate` to automatically generate a TypeScript declaration file for `route-parser`. The tool downloads the NPM package, runs the examples extracted from its documentation, gathers run-time information, and generates a TypeScript declaration file. The result is shown in Figure 1 and it is ready for use in a TypeScript project. For example, Visual Studio's code completion runs properly with it (Figure 1). If the `route-parser` package gets modified in the future, a new declaration file can be generated automatically using `dts-generate`. Our comparator tool can compare the new file for incompatibilities with the previous declaration file.

After filling in some background information on declaration files, Section 4 examines each step in the generation process in detail and refer back to this example for concreteness.

3 TypeScript Declaration Files

The declaration file shown in Figure 1 describes a package with a single exported class `Route`. The description of the class comprises the type signature of the constructor (line 4) and the methods `match` and `reverse` (lines 5 and 6).

!!!TODO: explain meaning of namespace.

This file is an instance of one of the standard templates for writing declaration files[4]: **module**, **module-class** and **module-function**. Each template corresponds to a different way of describing the exports of a JavaScript library. Choosing the template depends on the particular organization of the underlying JavaScript library:

module several exported functions,
module-class a class-like structure,
module-function exactly one exported functions.

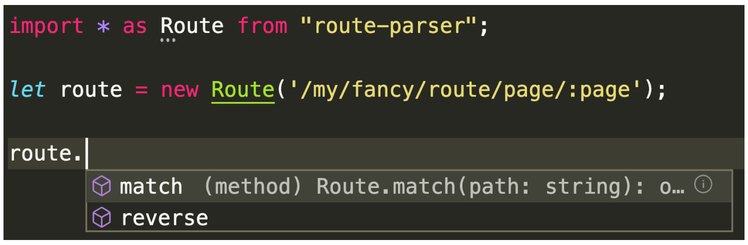
!!!TODO: example for simple **module**

[2] https://www.npmjs.com/package/route-parser
[4] https://www.typescriptlang.org/docs/handbook/declaration-files/templates.html

```
export = Route;

declare class Route {
  constructor(spec: string);
  match(path: string): object;
  reverse(params: object): string;
}

declare namespace Route {
}
```



```
import * as Route from "route-parser";

let route = new Route('/my/fancy/route/page/:page');

route.
```

match (method) Route.match(path: string): o…
reverse

Figure 1 **Declaration file for `route-parser` generated with `dts-generate`** - Constructor and methods are correctly identified. Declaration file can be correctly used in Visual Studio Code[3].

Listing 2 Example for template **module-function**

```
export = Abs;

declare function Abs(input: string): string;
```

The `route-parser` library is an instance of a **module-class**. The `abs` library provides a single function `Abs` so it uses template **module-function** as shown in 2.

4 The Generation of TypeScript Declaration Files

This section gives an overview of our approach to generating TypeScript declaration files from JavaScript libraries packaged in NPM. Figure 2 gives a rough picture of the inner working of our tool `dts-generate`. The input is an NPM package and the output is a TypeScript declaration file for the package if it is "sufficiently documented", which we substantiate in the next subsection.

As `dts-generate` is based on run-time information, exemplary code fragments that execute the JavaScript library are needed to obtain significant run-time information from running the instrumented library code.

The examples and the code base of the library are instrumented with Jalangi [9] to gather data flow information and type information at runtime. Jalangi is a configurable framework for dynamic analysis of JavaScript. It provides several analysis modules that we extended as needed to retrieve the required run-time information, which is then saved in a JSON file.

A second independent block uses the run-time information to generate a TypeScript declaration file. It infers the overall structure of the JavaScript library, its interfaces, and the types from the run-time information. The resulting declaration file is ready for use in the development process. Its content mimics the usage of the library in the example code fragments and matches the structure of the JavaScript library under analysis, so that the JavaScript code generated after compiling the TypeScript code runs without interface-related

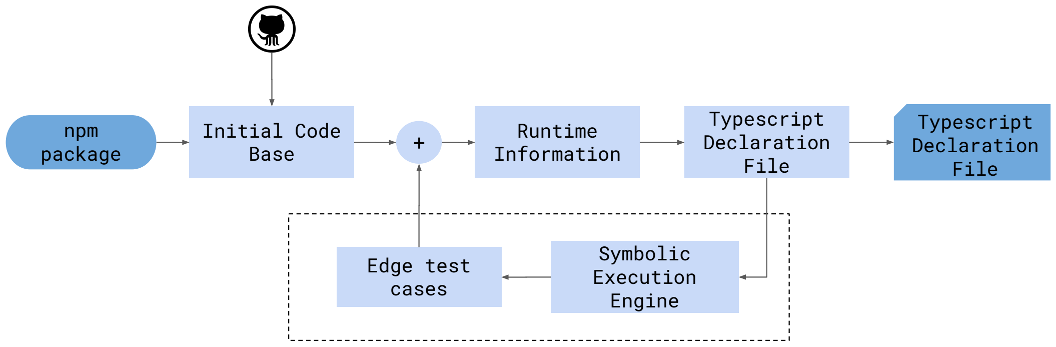

npm package
Initial Code Base
+
Runtime Information
Typescript Declaration File
Typescript Declaration File
Edge test cases
Symbolic Execution Engine

Figure 2 dts-generate - Architecture overview - Initial code base is retrieved from the NPM package's repository. A valid TypeScript Declaration File is generated using run-time information. A Symbolic Execution Engine creates test cases based on the generated Declaration File and via a feedback loop enriches the code base until the stopping criteria is reached. The final TypeScript Declaration File gets returned. Feedback loop through the Symbolic Execution Engine was not implemented. It can be added in a future to the existing architecture, without modifying the working blocks.

errors.

The command line interface is inspired by the `dts-gen` package [2]. Listing 2 shows that invoking the package is very simple and the only required argument is the name of the module published to the NPM registry.

4.1 Initial Code Base

To extract run-time information from a JavaScript library, it is necessary, by definition, to actually execute the code, because the analysis modules provided by Jalangi to gather information are only triggered if the instrumented code gets executed.

There are several options to obtain code fragments that drive the library code:

1. execute code that imports the library;
2. execute the test cases that come with the library;
3. execute code fragments extracted from the library documentation.

Option 1 does not solve our problem, it just delegates it to the importing library, which also needs code to drive it. Moreover, it is costly to download and instrument another package.

We considered option 2 under the assumption that most libraries would come with test cases. However, there is no standard for testing JavaScript code so that test cases were difficult to reap from the NPM packages: they use different directory structures, employ differing (or no) testing tools, or do not have tests at all.

In the end, option 3 was the most viable even though there is no standard for documentation, either. However, almost all repositories contain README files where the library authors briefly describe in prose what the code does, which problem it solves, how to install the application, how to build the code, etc. It is very common that developers provide code examples in the README files to show how the library works and how to use it. This observation holds in particular for NPM packages, which are generally created to solve a specific problem of JavaScript development.

Obtaining code examples for a specific NPM package is done in three steps.

- Obtain the repository's URL with the command `npm view <PACKAGE> repository.url`
- Retrieve the README file from the top-level directory of the repository.
- Extract the code examples from the README file. To this end, observe that README files are written using Markdown[5], a popular markup language. In such a file, it is customary to write code examples in code blocks labeled with the programming language, so that syntax highlighting is done correctly. Hence, we retrieve the code examples from code blocks labeled `js` or `javascript`, which both stand for JavaScript in Markdown.

For example, in the case of `route-parser` ...

Obtaining the code fragments from the examples provided in the README files of the repository proved to be an appropriate and pragmatic way of extracting the developer's intention and providing a useful initial code base with meaningful examples, thus avoiding possible cold start problems.

4.2 Run-time Information Gathering

The Runtime Information block described in Figure 2 gathers information such as:

- Function `f` was invoked where parameter `a` held a value of type `string` and `b` a value of type `number`.
- Property `foo` of parameter `a` of function `f` was accessed within the function.
- Parameter `a` of function `f` was used as operand for operator `==`.

The dynamic analysis framework Jalangi is used for gathering this kind of information. The configurable analysis modules enable programming custom callbacks that get triggered with virtually any JavaScript event. Our instrumentation observes the following events:

- Binary operations, like `==`, `+` or `===`.
- Variable declaration.
- Function, method, or constructor invocation.
- Access to an object's property.
- Unary operations, like `!` or `typeof`.

The implementation stores these observations as entities called `interactions`. They are used for translating, modifying, and aggregating Jalangi's raw event information to get an application specific data representation. The run-time information is saved as a JSON file that can be used for later processing. The tool is written in JavaScript and runs in NodeJS in a Docker container.

4.3 TypeScript Declaration File Generation

The next step in the pipeline after gathering the run-time information is the actual generation of the declaration file (cf. Figure 2). It is a lightweight, simple and fast application, which solely relies on the run-time information gathered in the previous step.

!!!TODO too vague: more information — how do you choose between the templates?

We use different fields (which?) from the run-time information to detect how the module is used to choose the most suitable template. The implemented templates are **module**, **module-class** and **module-function**.

[5] `https://www.markdownguide.org`

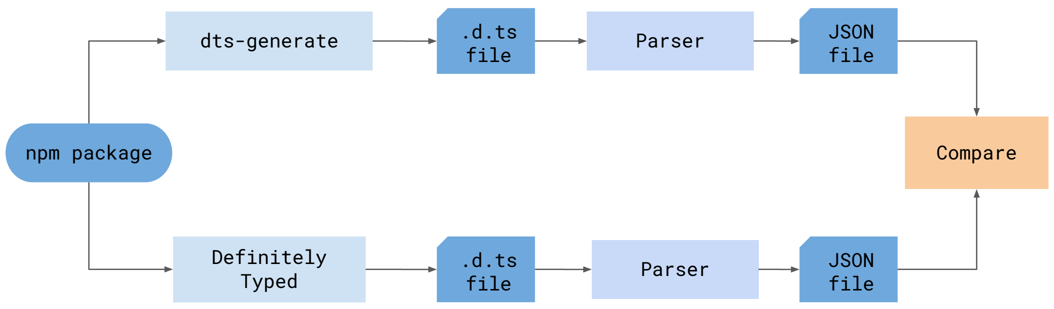


Figure 3 Evaluation of generated declaration files against DefinitelyTyped Repository - A parser transforms the generated declaration file and the equivalent file in the DefinitelyTyped repository into a JSON file using the TypeScript Compiler API [3]. Comparison is then performed on the JSON files, i.e. not on the declaration files.

Finally, interfaces are created by exploring `getField` and `methodCall` interactions from the run-time information. We gather the interactions for a specific argument and build the interface by incrementally adding new properties. Interactions within the `followingInteractions` field are recursively traversed, building a new interface in each recursion level.

!!!TODO: insufficient explanation of `followingInteractions` — needs to be mentioned in the previous step

5 Evaluation

After generating a declaration file for an NPM package, we need to evaluate its quality. To this end, we compare the generated declaration file against the one uploaded to the DefinitelyTyped repository for the same module, as shown in Figure 3. While this approach does not provide an absolute measure of quality, it gives us at least an indication of the pragmatics of `dts-generate`: The files on DefinitelyTyped are perceived to be useful by the community. If the accuracy of the generated files is comparable with the accuracy of the files on DefinitelyTyped, then `dts-generate` is a viable alternative to manually creating a definition file.

!!!TODO: check how many files on DT are just generated by `dts-gen` (e.g., the TypeScript tool)

To this end, we need a means of comparing two declaration files. Of course, we are not interested in a textual comparison, but in a comparison of the structures described by the files. Technically, our comparison of two structures A and B yields one of five outcomes:

- A and B are equivalent;
- A is more general than B;
- B is more general than A;
- A and B have a common generalization (i.e., they are compatible);
- none of the above holds (i.e., A and B are incompatible).

The first step is to parse declaration files, which can be achieved by using the TypeScript Compiler API, a library traverse the Abstract Syntax Tree of a TypeScript program [3]. The step also performs a sanity check of the generated declaration files as it rejects files with syntactic or semantics errors. The output of the parser is a structure where declared interfaces, functions, classes and namespaces are stored separately. Function arguments are correctly described, identifying complex types like union types or callbacks. Optional parameters are also identified. For classes, a distinction between the constructor and methods is made.

It remains to compare the parsed representations of two declaration files. As an example, we discuss the implementation of comparison for the **module-function** template.

The following criteria were applied:

- Number of declared functions: Checks the number of declared and exported functions for each of both declaration files.
- Name of declared functions: Checks whether both of the declaration files declared a function with the same name.
- Number of parameters: Checks the number of parameters of the declared functions. This is checked for optional and non-optional parameters.
  !!!TODO: what about parameter names? what about object types/interface types of parameters?
- Interfaces: Checks the number of declared interfaces and the fields within those interfaces.
- Errors: Indicates whether there are errors in the declaration files.

!!!TODO: too informal. we need to define an ordering on interfaces etc.

6 Results

!!!TODO: first we need to say, what we want to achieve with the experimental evaluation

> The conducted experiments included tests that consisted of replacing a specific type definition from DefinitelyTyped [1] with the one generated in the experiments: TypeScript compilation was successful, the generated JavaScript code ran without errors and code intelligence features performed by IDEs like code completion worked as expected.

Declaration files were generated for existing modules uploaded to the NPM registry. The DefinitelyTyped repository was used as a benchmark. Each one of the generated files was compared against the corresponding declaration file already uploaded to the repository.

Figure 4 shows that a declaration file was generated for 244 modules out of 6029 modules. Samples of the generated declaration files for templates **module**, **module-class** and **module-function** are presented in Section 6.2 - Declaration Files Generation.

6.1 Code Examples

Retrieving the code examples for the JavaScript libraries proved to be a pragmatic way of driving the type gathering at run time. However, as shown in Figure 4, it was only possible to obtain working code examples for 2260 packages. The process of getting a valid code example for a module is divided in four stages:

- Extracting repository URL.
- Extracting README file.
- Extracting code examples from README files.
- Executing code examples and discarding failing ones.

The results obtained for each on of them are described in the following sections.

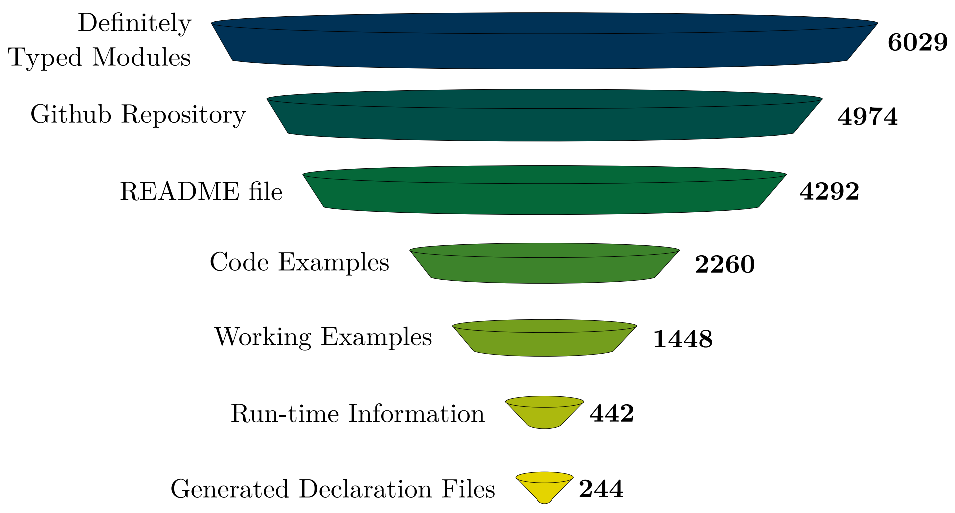


Figure 4 Number of analyzed modules for each stage of the experiment - A TypeScript Declaration File was generated for only 244 modules, out of 6029 modules in the DefinitelyTyped Repository. It was possible to gather valid run-time information for only 25% of the modules for which a Code Example was extracted.

Repository URL

The URL of the source repository could be retrieved for only 4974 packages. More than 1000 packages on NPM do not have a repository entry in their corresponding `package.json` file. Therefore, the `npm view <module> repository.url` command returns no value. Even important modules like `ace` provide no repository URL.

README Files

700 packages do not have a README file in their repository, although the implementation checks for several naming conventions like `readme.md` or `README.md`.

Extraction of Code Examples

In this step, we loose another 50% of modules! This loss is mainly explained because some developers do not wrap their code in a block using the `javascript` or `js` tags. However, as we are still left with code examples for 2200 modules, we did not further look into code extraction as this number was considered sufficient for evaluating the generation of declaration files.

Execution of Code Examples

We executed the remaining 2260 extracted code by installing the required packages and running the code as a NodeJS application. Unfortunately, the code examples only worked for 1448 modules. 812 modules did not run correctly and had to be discarded. Some failing samples were analyzed and there were mainly two reasons for the failure:

- The code fragment had been properly extracted but the code was not faulty. It was executing the library in an unsupported (obsolete?) way, which lead to a run-time error.
- The extracted code fragment was not intended to be executed and/or it was not even valid JavaScript code.

```
export = Abs;

declare function Abs(input: string
    ): string;
```

(a) abs/index.d.ts - Generated

```
declare function Abs(input: string
    ): string;
export = Abs;
```

(b) abs/index.d.ts - DefinitelyTyped

```
export = DirnameRegex;

declare function DirnameRegex():
    RegExp;
```

(c) dirname-regex/index.d.ts - Generated

```
export = dirnameRegex;

declare function dirnameRegex():
    RegExp;
```

(d) dirname-regex/index.d.ts - DefinitelyTyped

```
export = EscapeHtml;

declare function EscapeHtml(string
    : string): string;
```

(e) escape-html/index.d.ts - Generated

```
declare function escapeHTML(text:
    string): string;
declare namespace escapeHTML { }

export = escapeHTML;
```

(f) escape-html/index.d.ts - DefinitelyTyped

Figure 5 Results for **module-function** template

!!!TODO: the remaining two stages "run-time information" and "generated declaration files" need to be explained

6.2 Declaration Files Generation

This section exhibits some samples of the 244 generated declaration files. It shows some results for each of the implemented templates: **module**, **module-function** and **module-class**.

Figure 5 shows the generated declaration files for simple modules like `abs`, `dirname-regex`, and `escape-html`. All of them were generated using the **module-function** template. There are no differences between the generated files and the corresponding declaration files on DefinitelyTyped.

The left side of the figure shows the generated declaration file with `dts-generate`, the right side shows the corresponding file in the DefinitelyTyped repository. Functions are correctly detected; input and output types are accurately inferred.

!!!TODO: explain, what about

- uppercase/lowercase in dirname-regex?
- extra namespace in escape-html?

Templates of type **module-class** are shown for modules `flake-idgen`, `route-parser`, and `timer-machine` in Figure 6 and Figure 7, respectively. Properties of interfaces and class methods are correctly generated. Optional parameters are not detected, as it was not considered for the implementation.

For `flake-idgen`, the parameters of interface `ConstructorOptions` are correctly detected. The name of the interface differs because it is automatically generated based on the name of the argument variable. Optional properties were not implemented, which explains the type `undefined` for some properties. Analogously, the callback `cb` is inferred as `undefined`.

!!!TODO:

- why is the callback undefined? is it due to the example? (what does it look like?) why is it called `cb`?

```
export = FlakeIdgen;

declare class FlakeIdgen {
    constructor(options:
        FlakeIdgen.I__options);
    next(cb: undefined): Buffer;
}

declare namespace FlakeIdgen {
    export interface I__options {
        'id': undefined;
        'datacenter': number;
        'worker': number;
        'epoch': undefined;
        'seqMask': undefined;
    }
}
```

(a) flake-idgen/index.d.ts - Generated

```
interface ConstructorOptions {
    datacenter?: number;
    worker?: number;
    id?: number;
    epoch?: number;
    seqMask?: number;
}

declare namespace FlakeId { }

declare class FlakeId {
    constructor(options?:
        ConstructorOptions);
    next(callback?: (err: Error,
        id: Buffer) => void):
        Buffer;
}

export = FlakeId;
```

(b) flake-idgen/index.d.ts - DefinitelyTyped

Figure 6 Results for **module-class** module `flake-idgen`

- the export / class is named differently; doesn't that matter?

For `timer-machine`, the parameter `started` is inferred as `undefined` instead of marking it as optional. Methods that were not executed cannot appear in the generated declaration file. For that reason, a type corresponding to `TimerEvent` is not generated, either.

Finally, a sample for the **module** template for the `is-uuid` module is shown in Figure 8. Again, methods that were not executed by the extracted examples are not included in the declaration file.

It is worth mentioning that for some libraries the declaration file in DefinitelyTyped was not correct. For example, for the module `datadog-metrics`, some properties of an interface were included in the generated declaration file but they were not present in the declaration presented in DefinitelyTyped. However, as shown in Figure 9, the properties `aggregator` and `reporter` are not present in the DefinitelyTyped version, but they appear in the generated declaration file. However, they are indeed used by the library, as exposed in lines 2 and 3 of the library's source code shown in Figure 9c.

6.3 Evaluation

!!!TODO: explain what's going on in this subsection

!!!TODO: what's the baseline for the percentages in the paragraph, the 244 modules considered or all of DT? Ah, you say that in the figure; but then the figure is comparing apples with pears (percentages for different baselines!).

As shown in Figure 10, 20% of the declaration files in DefinitelyTyped are written using the **module-function**. However, 57% of the 244 generated declaration files are written with the **module-function** template. Additionally, the complexity of evaluating declaration files written with the **module-function** is considerably lower than for other templates. The evaluation for templates **module-class** and **module** was not implemented.

33 out of 116 evaluated modules have no difference with their corresponding declaration file in DefinitelyTyped.

!!!TODO: that's a bit lame as a result. See above for a more meaningful output.

```
export = Timer;

declare class Timer {
    constructor(start: undefined);
    start(): boolean;
    isStopped(): boolean;
    emit(): boolean;
    stop(): boolean;
    isStarted(): boolean;
    timeFromStart(): number;
    time(): number;
}

declare namespace Timer {
}
```

(a) timer-machine/index.d.ts - Generated

```
export as namespace Timer;
export = Timer;

declare namespace Timer {
    type TimerEvent = "start" | "
        stop" | "time";
}

declare class Timer {
    static get(reference: string):
         Timer;
    static destroy(reference:
        string): Timer;

    constructor(started?: boolean)
        ;

    isStarted(): boolean;
    isStopped(): boolean;
    start(): void;
    timeFromStart(): number;
    stop(): void;
    time(): number;
    toggle(): void;
    emitTime(): void;
    valueOf(): number;
    on(event: Timer.TimerEvent,
        callback?: () => void):
        void;
}
```

(b) timer-machine/index.d.ts - DefinitelyTyped

Figure 7 Results for **module-class** module `timer-machine`

```
export function v1(str: string):
    boolean;
export function v2(str: string):
    boolean;
export function v3(str: string):
    boolean;
export function v4(str: string):
    boolean;
export function v5(str: string):
    boolean;
```

(a) is-uuid/index.d.ts - Generated

```
export function v1(value: string):
     boolean;
export function v2(value: string):
     boolean;
export function v3(value: string):
     boolean;
export function v4(value: string):
     boolean;
export function v5(value: string):
     boolean;
export function nil(value: string)
    : boolean;
export function anyNonNil(value:
    string): boolean;
```

(b) is-uuid/index.d.ts - DefinitelyTyped

Figure 8 Results for **module** module `is-uuid`

```
export interface I__opts {
    'aggregator': undefined;
    'defaultTags': Array<any>;
    'reporter': undefined;
    'apiKey': string;
    'appKey': undefined;
    'agent': undefined;
    'host': string;
    'prefix': string;
    'flushIntervalSeconds': number
        ;
}

export class BufferedMetricsLogger
    {
    constructor(opts: I__opts);
    // ...
}
```

(a) datadog-metrics/index.d.ts - Generated

```
export interface LoggerOptions {
    apiKey?: string;
    appKey?: string;
    defaultTags?: string[];
    flushIntervalSeconds?: number;
    host?: string;
    prefix?: string;
}

export class BufferedMetricsLogger
     {
    constructor(
        options: LoggerOptions
    );
    // ...
}
```

(b) datadog-metrics/index.d.ts - Definitely-Typed

```
function BufferedMetricsLogger(opts) {
    this.aggregator = opts.aggregator || new Aggregator(
       opts.defaultTags);
    this.reporter = opts.reporter || new DataDogReporter(
       opts.apiKey, opts.appKey, opts.agent);
    this.host = opts.host;
    this.prefix = opts.prefix || '';
    this.flushIntervalSeconds = opts.flushIntervalSeconds;

    // ...
    // ...
}
```

(c) datadog-metrics/logger.js

Figure 9 Missing properties for module `datadog-metrics`

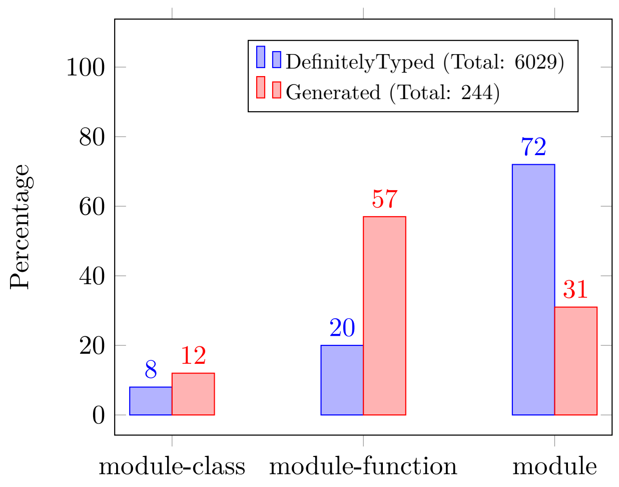


Figure 10 TypeScript templates distribution | Generated & DefinitelyTyped - Out of a total of 6029, 72% of the modules uploaded to the DefinitelyTyped repository use the **module** template and only 20% use the **module-function** one. However, 57% of the 244 generated declaration files use the **module-function** template.

Listing 3 Microsoft's dts-gen example - A declaration file for module `abs` is generated. Types are inferred as `any`. The correct `module-function` template is used.

```
$ npm i -g dts-gen
$ npm i -g abs
$ dts-gen -m abs
Wrote 5 lines to abs.d.ts.

$ cat abs.d.ts
/** Declaration file generated by dts-gen */

export = abs;

declare function abs(input: any): any;
```

7 Related Work

Microsoft's dts-gen

Microsoft developed `dts-gen`, a tool that creates starter declaration files for JavaScript libraries [2]. Its documentation states that the result is however intended to be only used a starting point. The outcome needs to be refined afterwards by the developers.

The tool analyzes the shape of the objects at runtime after initialization without executing the library. This results in many variables being inferred as `any`. Listing 3 shows an example for module `abs`.

The solution presented in this work, however, is intended to generate declaration files that are ready to be uploaded to DefinitelyTyped without further manual intervention. Any amount of manual work that a developer needs to do on a declaration file after updating JavaScript code increases the risk for having discrepancies between the declaration file and the implementation.

Formal aspects like applying the right template and using the correct syntax are perfectly covered by `dts-gen`.

TSInfer & TSEvolve

TSInfer and TSEvolve are presented as part of TSTools [7]. Both tools are the continuation of TSCheck [4], a tool for looking for mismatches between a declaration file and an implementation.

TSInfer proceeds in a similar way than TSCheck. It initializes the library in a browser and it records a snapshot of the resulting state and then it performs a light weight static analysis on all the functions and objects stored in the snapshot.

The abstraction and the constraints they introduced as part of the static analysis tools for inferring the types have room for improvement. A run-time based approach like the one presented in our work will provide more accurate information, thus generating more precise declaration files.

Since they analyze the objects and functions stored in the snapshot, they faced the problem of including in the declaration file internal methods and properties that developers wanted to hide. Run-time information would have informed that the developer has no intention of exposing such methods.

Moreover, TSEvolve performs a differential analysis on the changes made to a JavaScript library in order to determine intentional discrepancies between declaration files of two consecutive versions. We consider that a differential analysis may not be needed. If the developer's intention is accurately extracted and the execution code clearly represents that intention then the generated declaration file would already describe the newer version of a library without the need of a differential analysis.

TSTest

TSTest is a tool that checks for mismatches between a declaration file and a JavaScript implementation [8]. It applies feedback-directed random testing for generating type test scripts. These scripts will execute the library in order to check if it behaves the way it is described in the declaration file. TSTest also provides concrete executions for mismatches.

We evaluated the generated declaration files comparing them to the declaration files uploaded to DefinitelyTyped. The disadvantage of doing this is that since the uploaded files are written manually, they could already contain mismatches with the JavaScript implementation. However, it is a suitable choice for a development stage since it is used as a baseline.

In a final stage, declaration files need to be checked against the proper JavaScript implementation and TSTest has to be definitely taken into account.

8 Conclusion

We have presented `dts-generate`, a tool for generating a TypeScript declaration file for a specific JavaScript library. The tool downloads code samples written by the developers from the library's repository. It uses these samples to execute the library and gather data flow and type information. The tool finally generates a TypeScript declaration file based on the information gathered at run-time.

We developed an architecture that supports the automatic generation of declaration files for specific JavaScript libraries without additional manual tasks. The architecture contemplates a future incorporation of a Symbolic Execution Engine that refines the initial code base enabling the exploration of new execution paths. However not implemented in this work, its incorporation would result in small incremental modifications to the presented architecture as it is considered to only expand the existing code base.

Building an end-to-end solution for the generation of TypeScript declaration files was prioritized over type inference accuracy. Consequently, types were taken over from the values at run-time. Since developers expose through code how a library should be used, obtaining the types from the code examples extracted from the repositories proved to be a pragmatic and effective approximation, enabling to work on specific aspects regarding the TypeScript declaration file generation itself.

We built a mechanism to automatically create declaration files for potentially every module uploaded to DefinitelyTyped. We managed to generate declaration files for 244 modules. We compared the results against the corresponding files uploaded to DefinitelyTyped by creating a TypeScript declaration files parser and a comparator.

We exposed the fundamental aspect of capturing the developer's intention when inferring types in JavaScript. Instead of applying constraints and restrictions for operations with certain types, we presented a proposal where common practices are favored. Uncommon usage is not forbidden but greatly disfavored. Accordingly, we collected evidence regarding the usage of JavaScript operators by analyzing 400 libraries.

Finally, the architecture is composed of different blocks that interact with each other. Each block is independent and has a well defined behavior as well as clear input and output values. As a result, each block can be independently and simultaneously improved.

References

1 Definitelytyped. `http://definitelytyped.org/`.

2 dts-gen: A typescript definition file generator. `https://github.com/microsoft/dts-gen`.

3 Typescript compiler api. `https://github.com/microsoft/TypeScript/wiki/Using-the-Compiler-API`.

4 Asger Feldthaus and Anders Møller. Checking correctness of TypeScript interfaces for JavaScript libraries. In Andrew P. Black and Todd D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 1–16. ACM, 2014. `doi:10.1145/2660193.2660215`.

5 Zheng Gao, Christian Bird, and Earl T. Barr. To type or not to type: Quantifying detectable bugs in JavaScript. In Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard, editors, *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 758–769. IEEE / ACM, 2017. `doi:10.1109/ICSE.2017.75`.

6 Github statistics. `https://madnight.github.io/githut`.

7 Erik Krogh Kristensen and Anders Møller. Inference and evolution of TypeScript declaration files. In Marieke Huisman and Julia Rubin, editors, *Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10202 of *Lecture Notes in Computer Science*, pages 99–115. Springer, 2017. `doi:10.1007/978-3-662-54494-5\_6`.

8 Erik Krogh Kristensen and Anders Møller. Type test scripts for TypeScript testing. *PACMPL*, 1(OOPSLA):90:1–90:25, 2017. `doi:10.1145/3133914`.

9 Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In Bertrand Meyer, Luciano Baresi, and Mira Mezini, editors, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 488–498. ACM, 2013. `doi:10.1145/2491411.2491447`.

10 Typescript language specification. `https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md`.