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 complex and large applications in TypeScript, a typed dialect of JavaScript. TypeScript applications integrate JavaScript libraries via typed descriptions of their APIs called declaration files. These files are available in public repository DefinitelyTyped. This repository is maintained manually, 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 against the declaration file provided on DefinitelyTyped, the standard public repository for declaration files. 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 [12]. It is also gaining popularity for back-end applications running in NodeJS. Its dynamic typing speeds up programming enabling developers to create simple pieces of code very fast, making JavaScript a very appealing programming language.

JavaScript is being used for creating complex and large applications. However, JavaScript was not intended to be more than a scripting language. Maintaining and evolving large JavaScript codebases is notably challenging. Mistakes such as mistyped property names and misunderstood or unexpected type coercion cause developers to spend a significant amount of time in debugging sessions. A JavaScript code blog[1] compiles experiences from developers facing unexpected situations while programming in JavaScript. Listing 1 exposes some of these unintuitive JavaScript behaviors.

The overhead produced by such dynamic typing is not present in other languages that use build tools based on type information. The situation motivated the creation of TypeScript, a superset of JavaScript with typed annotations [11]. It has become a widely used alternative among JavaScript developers, since it incorporates features that are helpful for developing and maintaining large applications [5]. TypeScript enables the early detection of run-time

[1] https://wtfjs.com

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
"01" < "00100"; // false
[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
```

errors detection and the integration of code intelligence tools like auto-completion in the IDEs.

Existing JavaScript libraries can be used in a TypeScript project by adding a declaration file that contains a typed description of the library's API. Declaration files are stored in a repository called DefinitelyTyped that contains declaration files for more than 6000 JavaScript libraries [2]. Unfortunately, declaration files need to be manually created and maintained, which is error prone and time consuming. TypeScript does not perform any run-time check on these declaration files. A discrepancy between the declaration file and its corresponding JavaScript library would lead to additional frustration and debugging sessions, since type checks and code-intelligence features would be inaccurate.

Some previous work tackled the problem of automatically searching for mismatches between a declaration file and implementation code [4]. TSTest adapted the feedback-directed random testing technique, mainly used for testing Java libraries, to detect discrepancies between a declaration file and a JavaScript library [7]. Tools like TSInfer and TSEvolve are designed for assisting the creation of new declaration files and supporting the evolution the declaration file when the corresponding JavaScript library gets modified [6], respectively. TypeScript itself developed `dts-gen`, a tool that generates a declaration file that is meant to be used only as a 'starting point for writing a high-quality declaration file' [3].

We explore in this work the possibilities for improving the existing tools. We present the tool `dts-generate`. It is based on an architecture that supports the generation of declaration files for an existing JavaScript library published to the NPM registry. The tool will gather data flow and type information at run-time and generate a declaration file based on that information.

The architecture supports the future incorporation of a Symbolic Execution Engine that expands the initial code base using the signatures in the declaration file. The iterative process of exploring new execution paths will refine the generated declaration file in each iteration.

Finally, we generated declaration files for 244 JavaScript libraries and evaluated them against DefinitelyTyped.

The contributions of this paper are as follows.

- We introduce an architecture that supports generating TypeScript Declaration Files for a given JavaScript Library using run-time information. The architecture supports the future incorporation of a Symbolic Execution Engine that expands the initial code base using the signatures in the declaration file. The iterative process of exploring new execution paths will refine the generated declaration file in each iteration.

```
export = Route;

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

declare namespace Route {
}
```

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].

- We present the tool `dts-generate`, a command line application that generates a valid TypeScript Declaration File for a specific NPM package using run-time information.
- Next, we deliver a test framework that supports generating a TypeScript Declaration File using `dts-generate` for all JavaScript libraries in the DefinitelyTyped Repository. We present the results of running `dts-generate` for 244 of these libraries.
- Finally, we present a TypeScript Declaration Files parser and comparator. Both tools were necessary for evaluating the results against DefinitelyTyped.

2 Motivating Example

The NPM package `route-parser` is a simple route parsing, matching, and reversing library for Javascript[2]. It has about 35000 weekly downloads and 221 NPM packages depend on it. If a developer is creating or extending a JavaScript library written in TypeScript that depends on `route-parser`, the TypeScript compiler and IDEs will need a declaration file for that JS library in order to perform static checking and code completion, respectively. We use `dts-generate` to automatically generate a TypeScript Declaration File for this library. The tool will download the npm package, run it, gather run-time information and generate a valid TypeScript Declaration File. Figure 1 shows the generated declaration file for package `route-parser` using `dts-generate`. The result is a valid declaration file that can be used in a TypeScript project. For example, it can be seen in Figure 1 that Visual Studio Code performs code completion properly. Finally, if `route-parser` gets modified in a future, a new declaration file could be automatically generated using `dts-generate`.

[2] https://www.npmjs.com/package/route-parser

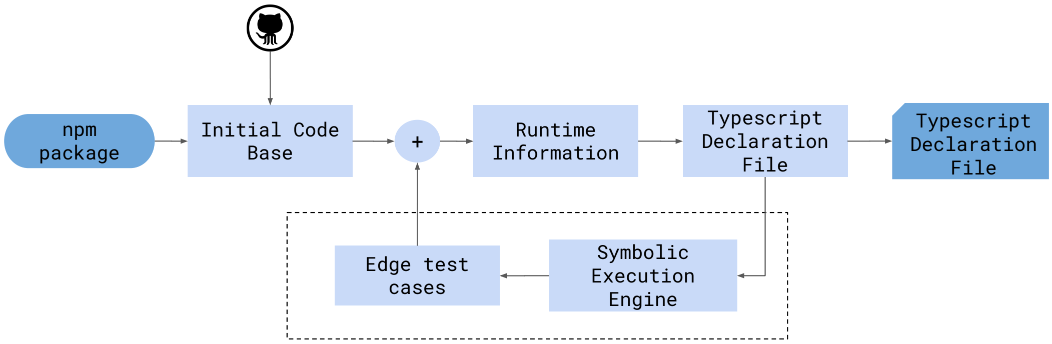

Figure 2 dts-generate - Architecture overview - Initial code base is retrieved from the npm package's repository. A valid TypeScript Declaration File is generated using run-time information. A Symbolic Execution Engine creates test cases based on the generated Declaration File and via a feedback loop 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.

3 dts-generate - Generation of TypeScript Declaration Files

We introduce `dts-generate`, a command line tool which generates a valid TypeScript Declaration File for a specific JavaScript Library uploaded to the NPM Registry, as explained in Figure 2. The tool is intended to be used on existing, published `npm` packages. The generated output TypeScript declaration file is a valid file which can be used for development and uploaded to the DefinitelyTyped Repository.

Code examples that execute the JavaScript Library are needed in order to extract the run-time information via code instrumentation. It is achieved by retrieving the examples provided in the README files of the repositories of the different libraries. This is generally the place where developers explicitly show how to use their code. It showed to be an appropriate and pragmatic way of extracting the developer's intention and providing an useful initial code base with meaningful examples, thus avoiding a possible cold start problem.

The examples and the code base of the library are instrumented with Jalangi [9][10] to gather data flow information and type information at runtime. Jalangi is a dynamic analysis configurable framework that provides several analysis modules that were extended as needed to retrieve the required run-time information, which is then saved to an output JSON file.

A second independent block uses the run-time information to generate a TypeScript Declaration File. It infers the overall structure of the JS Library, the interfaces and the types from the JSON file.

The declaration file returned by the method is valid and fully functional, making it suitable for being used within the development process. It contains no errors and matches the structure of the JavaScript Library under analysis, so that the JavaScript code generated after compiling the TypeScript code runs without errors. The conducted experiments included tests that consisted on replacing a specific type definition from DefinitelyTyped [2] 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.

The command line interface was inspired in the `dts-gen` [3] package. It can be seen in Listing 2 that invoking the package is very simple and the only required argument is the

Listing 2 dts-generate usage - Example of how to generate a declaration file for module `abs`.

```
$ ./dts-generate abs
$ cat output/abs/index.d.ts
export = Abs;

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

name of the module published to the npm registry.

3.1 Initial Code Base

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

It was decided to extract the code examples that execute the JavaScript Library from the Readme files of the repository associated to the NPM Package. Readme files are usually used by developers for briefly describing what the code does, what 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 code works and how to use it. This is specially true for NPM Packages, which are in general created to solve a specific problem of JavaScript development.

Obtaining code examples for a specific NPM Package is achieved in three steps:

- Obtain the repository url from the package: The command `npm view <PACKAGE> repository .url` can be used for retrieving the url of the package's repository.
- Retrieve the README file from the repository.
- Extract the code examples from the README file: Readme files are written using Markdown[4], a very common lightweight and simple markup language. It is very common to write code examples within code blocks indicating the programming language, so that it gets highlighted with the specific syntax. The Markdown identifiers for JavaScript are `js` or `javascript`. The code examples are finally retrieved by filtering the content within the corresponding code blocks.

3.2 Run-time Information Gathering

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

- Function `f` got invoked with parameters `a` and `b` with types `string` and `number`.
- Property or method `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 used for gathering this kind of information is Jalangi. The configurable analysis modules enable programming custom callbacks that get triggered with virtually any JavaScript event. The events that are observed are:

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

[4] https://www.markdownguide.org

- 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 in order to get an application specific data representation. The run-time information is finally returned as a JSON file that can be used for later processing. The tool is written in JavaScript and runs in Node.js within a Docker container.

3.3 TypeScript Declaration File Generation

Overview

The actual generation of the declaration file is the next step in the pipeline after gathering the run-time information, as shown in Figure 2. It is a lightweight, simple and fast application, which does not interact with the actual JavaScript module at run-time. Instead, it uses the JSON output file containing the run-time information and generates a TypeScript declaration file which is use ready to be used within a TypeScript project. The tool itself is written in TypeScript and runs within a Docker container in NodeJS.

Templates

TypeScript provides templates for writing declaration files[5] and each template corresponds to a different way of exporting a JavaScript module. The tool uses different fields from the run-time information to detect how the module is being used in order to choose the right templates accordingly. The implemented templates are `module`, `module-class` and `module-function`.

Interfaces

Finally, interfaces are created by exploring `getField` and `methodCall` interactions from the run-time information. The code will 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.

3.4 Evaluation

After generating a declaration file for a published NPM module, it is necessary to evaluate the quality of it. It was decided to compare the generated declaration file against the one uploaded to the DefinitelyTyped repository for the same module, as shown in Figure 3.

Parsing

Before comparison, declaration files need to be parsed so that both of them share the same structure. It was achieved by using the TypeScript Compiler API, a library developed by Microsoft that allows to traverse the Abstract Syntax Tree in an easy and intuitive way [1].

The parsing consists in creating a structure where declared `interfaces`, `functions`, `classes` and `namespaces` are stored separated. 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. Finally, syntax and semantic errors are also checked by the TypeScript Compiler API.

---

[5] https://www.typescriptlang.org/docs/handbook/declaration-files/templates.html

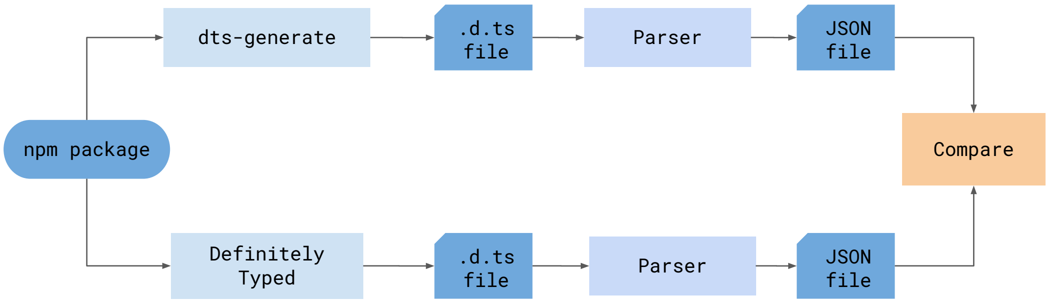

Figure 3 Evaluation of generated declaration files against DefinitelyTyped Repository - A parser transforms the generated declaration file and the equivalent file in the DefinitelyTyped repository into a JSON file using the TypeScript Compiler API [1]. Comparison is then performed on the JSON files, i.e. not on the declaration files.

The tool is called `parse-dts` and is naturally written in TypeScript. It also runs in NodeJS within its corresponding Docker container.

Comparator

An independent tool will compare two parsed declaration files. As described in Figure 3, the comparator will compare the generated declaration file against the corresponding file in the DefinitelyTyped repository.

It was discovered that the implementation was easier when focusing on each template independently. For this implementation, only the `module-function` was considered for the comparison.

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.
- Interfaces: Checks the number of declared interfaces and the fields within those interfaces.
- Errors: Indicates whether there are errors in the declaration files.

4 Results

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 4.2 - Declaration Files Generation.

4.1 Code Examples

Retrieving the code examples from the JavaScript libraries' repositories proved to be a pragmatic way of capturing the types. However, as shown in Figure 4, working code examples

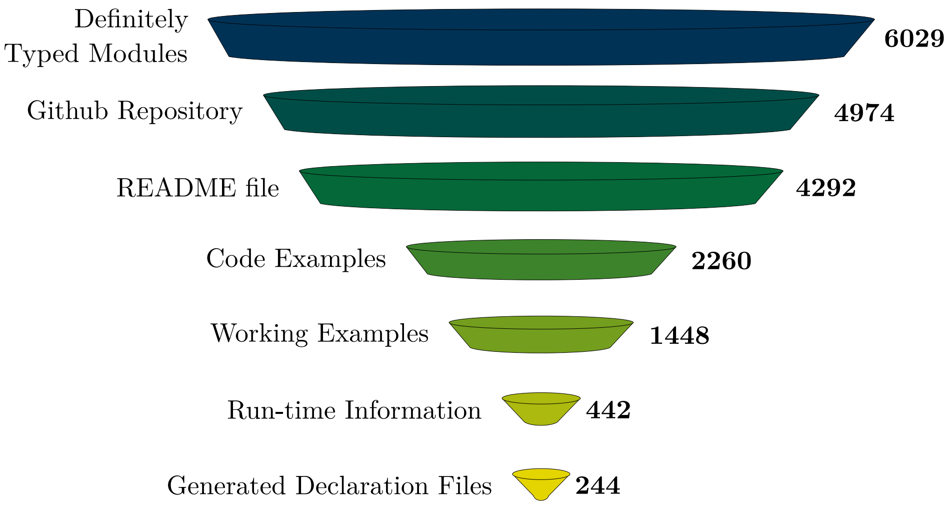

Figure 4 Number of analyzed modules for each stage of the experiment - A TypeScript Declaration File was generated for only 244 modules, out of 6029 modules in the DefinitelyTyped Repository. It was possible to gather valid run-time information for only 25% of the modules for which a Code Example was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

for only 2260 modules could be retrieved. The process of getting a valid code example for a module is divided in 4 blocks:

- Extracting repositories url.
- Extracting readme files.
- Extracting code examples within readme files.
- Executing code examples and discarding failing ones.

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

Repositories URL

The url of the repositories could be retrieved for only 4974 modules. More than 1000 modules do not have the repository entry in their corresponding `package.json` files. Therefore, the `npm view <module> repository.url` command returns an empty value. This is even happening for important modules like `ace`.

Readme Files

700 modules simply do not have a readme file in their repositories. The implementation does contemplate, however, different naming conventions like `readme.md` or `README.md`.

Code Examples Extraction

The 50% loss is mainly explained because developers did not wrap their code around a block using the `javascript` or `js` tags. Counting with code examples for 2200 modules was considered to be enough for evaluating the generation of declaration files.

Code Examples Execution

The 2260 extracted code examples were executed by installing the required packages and running the code as a `node` application. Working and functional code examples could only be

```
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 Module-function results - Results are shown for modules `abs`, `dirname-regex`, `escape-html`. On the left side the generated declaration file with `dts-generate`. On the right side the corresponding file in the DefinitelyTyped repository. Functions are correctly detected and input types are accurately inferred. Both files are parsed for comparison, as explained in Section 3.4 - Evaluation. Therefore, subtle differences in the syntax between both files are not important.

extracted for 1448 modules. 812 modules did not run correctly and were discarded. Some failing samples were analyzed and there were mainly two reasons for the failure:

1. The code example had been properly extracted but the code itself was not working. It was executing the library in an unsupported way, hence the error at run-time.
2. The extracted code example was not intended to be executed or it was not even valid JavaScript code.

4.2 Declaration Files Generation

The following 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 5f 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 uploaded to DefinitelyTyped.

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. Finally, `module` template is presented for `is-uuid` module in Figure 8.

It is worth mentioning that for some libraries the declaration file in DefinitelyTyped was not correct. For example, for `datadog-metrics`, some properties of an interface were included in the generated declaration file but they were not present in the one in DefinitelyTyped. However, as shown in Figure 9, the properties are indeed used in the source code and should be included.

```
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 Module-class results | flake-idgen - Parameters of interface `ConstructorOptions` are correctly detected. Name of interface differs since it is automatically generated based on the name of the argument variable. Optional properties were not implemented, hence the `undefined` type for some properties. Analogously, callback `cb` is inferred as `undefined`.

4.3 Evaluation

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.

5 Related Work

Microsoft's dts-gen

Microsoft developed `dts-gen`, a tool that creates starter declaration files for JavaScript libraries [3]. 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`.

```
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 Module-class results | timer-machine - Parameter `started` is inferred as `undefined` instead of marking it as optional. Methods that were not executed do not appear in the generated declaration file.

```
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 Module results | is-uuid - Methods that were not executed are not included in the declaration file.

```
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 - DefinitelyTyped

```
1  function BufferedMetricsLogger(opts) {
2      this.aggregator = opts.aggregator || new Aggregator(
           opts.defaultTags);
3      this.reporter = opts.reporter || new DataDogReporter(
           opts.apiKey, opts.appKey, opts.agent);
4      this.host = opts.host;
5      this.prefix = opts.prefix || '';
6      this.flushIntervalSeconds = opts.flushIntervalSeconds;
7
8      // ...
9      // ...
10 }
```

(c) datadog-metrics/logger.js

Figure 9 Missing properties | datadog-metrics - Properties `aggregator` and `reporter` are not 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 c.

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;
```

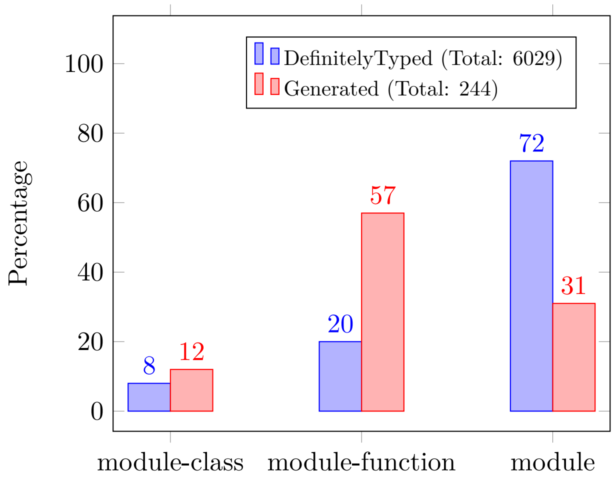

Figure 10 TypeScript templates distribution | Generated & DefinitelyTyped - Out of a total of 6029, 72% of the modules uploaded to the DefinitelyTyped repository use the `module` template and only 20% use the `module-function` one. However, 57% of the 244 generated declaration files use the `module-function`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

TSInfer & TSEvolve

TSInfer and TSEvolve are presented as part of TSTools [6]. 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 [7]. 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.

6 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 TypeScript Compiler API. *https://github.com/microsoft/TypeScript/wiki/Using-the-Compiler-API*.

2 DefinitelyTyped. *http://definitelytyped.org/*.

3 dts-gen: A TypeScript Definition File Generator. *https://github.com/microsoft/dts-gen*.

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 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.

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

8 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*. ACM, 2013. URL: http://dl.acm.org/citation.cfm?id=2491411.

9 Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for javascript. In Meyer et al. [8], pages 488–498. doi:10.1145/2491411.2491447.

10 Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. Jalangi: a tool framework for concolic testing, selective record-replay, and dynamic analysis of javascript. In Meyer et al. [8], pages 615–618. doi:10.1145/2491411.2494598.

11 TypeScript Language Specification. *https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md*.

12 Github Statistics. *https://madnight.github.io/githut*.