5ème année Informatique
2017-2018

Développement d'un outil analysant du code Angular (TypeScript)

Rapport de projet Test Logiciel

Étudiants :
Alexis BUSSENEAU
Grégoire LINOT
Mohamed RACHID
Adrien THÉBAULT

Encadrant :
Arnaud BLOUIN

Introduction

Pour le cours de Test Logiciel de 5ème année d'informatique à l'INSA de Rennes nous avons réalisé un analyseur de code Angular. Ce projet est divisé en plusieurs sous tâches, qui permettent de réaliser le projet pas à pas.

Dans ce rapport nous allons vous présenter le sujet puis nous vous présenterons ce qui a été fait et enfin nous exposerons les limites de notre projet ainsi que les difficultés rencontrées.

1 - Présentation

1.1 - Angular et les composants

Angular est un framework JavaScript libre et open-source développé par Google pour construire des applications client en HTML avec JavaScript ou TypeScript. Il sert notamment à dynamiser le HTML et permet la création de ses propres balises et attributs. Angular repose sur le patron de conception MV* ou MVW (Model-View-Whatever).

1.2 - TypeScript

TypeScript est un langage de programmation libre et open-source développé et maintenu par Microsoft. C'est un sur-ensemble syntaxique strict de JavaScript qui ajoute notamment un typage facultatif au langage. TypeScript peut être utilisé pour développer des applications JavaScript pour l'exécution côté client ou côté serveur (Node.js).

TypeScript est conçu pour le développement de grandes applications et compile en JavaScript. Comme TypeScript est un sur-ensemble de JavaScript, les programmes JavaScript existants sont également des programmes TypeScript valides. Le compilateur TypeScript est lui-même écrit en TypeScript et compilé en JavaScript.

1.3 - Objectif du projet

L'objectif du projet est de développer un outil permettant d'analyser statiquement une application Angular pour en extraire des informations dans un but de compréhension et pour éventuellement trouver des bugs.

2 - Travail réalisé

2.1 - Analyse des composants

Nous avons donc commencé par développer une fonctionnalité qui permettait d'identifier tous les composants d'une application. Pour réaliser cela, nous avons utilisé l'API du compilateur de TypeScript qui nous a permis, à partir d'une liste de fichiers, de créer une représentation abstraite d'un programme.

Au début nous utilisions une fonction de l'API du compilateur ***createSourceFile*** qui nous donnait un arbre syntaxique abstrait (AST) suffisant pour récupérer les composants mais insuffisant pour récupérer facilement les déclaration des attributs et des fonctions de classe. Nous nous sommes donc tournés vers la fonction ***getTypeChecker*** qui donne ces informations.

L'API du compilateur nous permet de récupérer toutes les déclarations de classes, de propriétés, de fonctions et d'objets littéraux. A partir de là, nous avons pu générer un arbre syntaxique abstrait et récupérer la liste de composants de notre application.

2.2 - Analyse des attributs et des méthodes utilisés

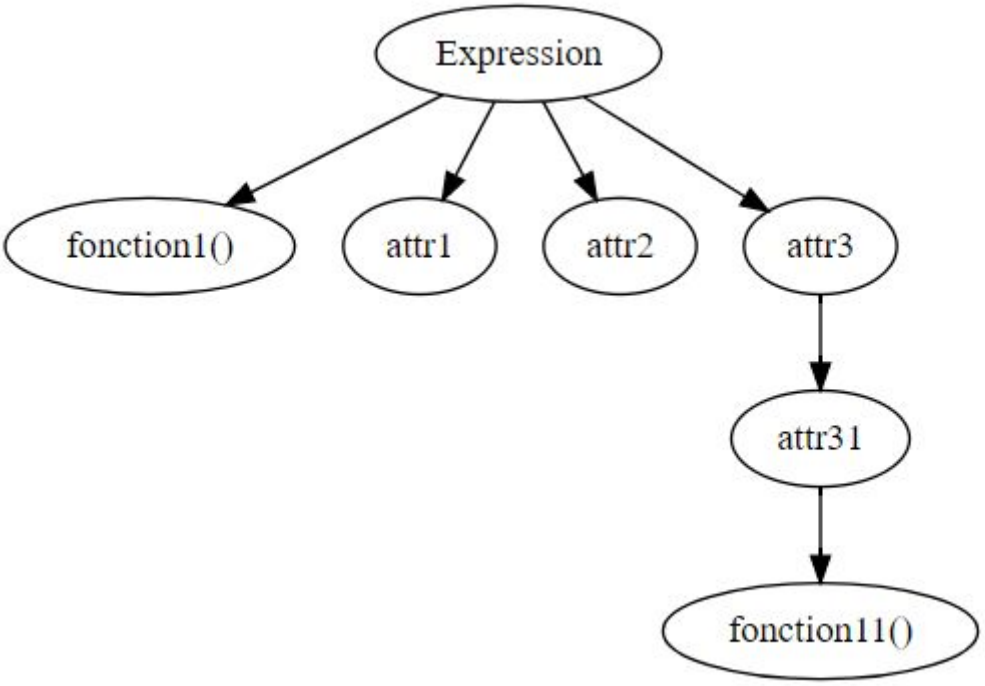
Grâce à notre arbre syntaxique abstrait, nous avons la liste des propriétés et des méthodes de chaque composant de notre application, nous avons donc ensuite analysé les templates des différents composants afin de faire la liste des attributs, méthodes et objets qui y sont utilisés.

Ainsi, en analysant complètement le DOM des différents templates, qu'ils soient définis comme une chaîne de caractères ou dans un fichier à part, nous avons pu extraire les attributs des balises contenant des expressions Angular en nous basant sur les règles suivantes :

1. Tout attribut d'un élément du DOM qui commence par ***ng, {{, [** ou **bind-target** est un binding Angular unidirectionnel allant du TypeScript vers la Vue
2. Tout attribut d'un élément du DOM qui commence par **(** ou **on-target** est un binding Angular unidirectionnel allant de la Vue vers le TypeScript (i.e. les évènements)
3. Tout attribut d'un élément du DOM qui commence par **[(,** ou **bindon-target** est un binding Angular bidirectionnel

A ces expressions là, nous avons également ajouté tous les bindings de type {{ maVariable }} qui se situent dans le texte. Ces bindings sont unidirectionnels allant du TypeScript vers la vue.

Une fois les expressions Angular récupérées, il nous a suffit de les analyser afin d'en ressortir la liste des objets, attributs et méthodes qui y sont utilisés. Par exemple l'expression `fonction1(attr1,attr2,attr3.attr31.fonction11()).fonction2()` donnera l'arbre syntaxique suivant :



On peut remarquer que `fonction2()` n'est pas présente dans l'arbre car on ne peut pas inférer le type de retour de la fonction `fonction1()`

Nous comparons ensuite cette dernière avec la liste obtenue lors de l'analyse des composants décrite en **2.1** afin de pouvoir détecter les erreurs.

Pour les ***ngFor**, qui permettent de parcourir un tableau en attribuant un nom de variable à l'élément courant du tableau, nous avons fait le choix de remplacer tous les appels à l'élément courant par un appel au premier élément du tableau du *ngFor. Par exemple, le code suivant :

```
<div *ngFor="let current of array">
    {{ current.name }}
</div>
```

sera analysé comme étant le code suivant :

```
<div>
    {{ array[0].name }}
</div>
```

3 - Difficultés rencontrées

3.1 - Manque de Documentation

La documentation de l'API du transpiler TypeScript est inexistante. Les seules manières de comprendre l'API est soit d'étudier les exemples fournis pour les cas simples, soit d'utiliser le débogueur et d'essayer de comprendre la structure des objets JavaScript en fonctions des paramètre d'entrée.

3.2 - Typage facultatif de TypeScript (**any**)

Le principal obstacle que nous avons rencontré est le typage facultatif de TypeScript. En effet, TypeScript permet de ne pas préciser le type des variables à la déclaration. Le type prend la valeur "any" par défaut, et peut correspondre à n'importe quel type (comme en JavaScript).

A cause de cela, notre solution ne peut pas prendre en compte les types des éléments dans les tableaux ou encore vérifier le type de retour des fonctions. Cela rend donc l'analyse statique difficile, il faudrait faire une vérification à l'exécution (analyse dynamique) pour que cela puisse fonctionner.

3.3 - Les éléments d'un tableau (*ngFor)

Cette difficulté découle directement de la précédente. En TypeScript, et donc en JavaScript, les tableaux peuvent contenir différents types d'objets à un instant donné, ainsi un tableau pourrait contenir comme premier élément un objet qui aurait une propriété "toto" qui n'existerait pas dans le second élément du même tableau. Par exemple :

```
export class AppComponent {
      title : 'Titre';
      elements : [{ toto: 42 }, { simo: "yoyo" }]
}

<h1>{{ title }}</h1>
<h2>Liste des éléments : </h2>
<ul>
      <li *ngFor="let element of elements">{{ element.toto }}</li>
</ul>
```

Cet exemple parfaitement fonctionnel démontre toute la difficulté de l'analyse des tableaux : l'appel à **element.toto** est vrai sur la moitié des éléments du tableau, nous ne pouvons donc pas lever d'erreur. Cependant, il est faux sur l'autre moitié des éléments du tableau, nous devrions donc en lever une.

Dans notre cas, le tableau est déclaré de manière statique. Il est donc aisé de l'analyser en amont. Cependant, il existe d'autres cas où le tableau est le résultat d'une exécution dynamique (par exemple une requête AJAX) qui est impossible de savoir avant l'exécution.

Qui plus est, le tableau est ici déclaré de manière statique, il est donc aisé de l'analyser mais nous pouvons imaginer un exemple où le tableau serait le résultat d'une requête AJAX qu'il serait donc impossible d'analyser avant l'exécution.

4 - Contributions

Pour la réalisation de ce projet, nous nous sommes répartis le travail de la manière suivante :

1. **Alexis BUSSENEAU** s'est occupé de la qualité du code (i.e. refactoring, cohérence du code, erreur de compilation)
2. **Grégoire LINOT** s'est occupé de la génération de l'arbre syntaxique abstrait
3. **Mohamed RACHID** s'est occupé de la gestion de projet et de l'analyse des problématiques
4. **Adrien THÉBAULT** s'est occupé de l'analyse des templates des différents composants de l'application

Conclusion

Malgré les difficultés rencontrées lors du développement de l'outil, le projet est fonctionnel et permet de relever des potentielles erreurs afin d'améliorer la qualité logicielle d'une application Angular. Il est a noter également que certaines consignes comme les *@Input* / *@Ouput* (fonctionnalités expérimentales) ou utiliser le .dot pour représenter la structure de l'application Angular n'étaient que des pistes de recherche qui nous ont permis de mieux visualiser le projet mais qui ne nous ont pas semblé intéressantes à implémenter.

Par ailleurs, Angular est un framework qui évolue rapidement et dont une nouvelle version majeure sort tous les six mois, de ce fait, notre analyseur est déjà obsolète puisque Angular 5.0 est sorti début Octobre, des fonctionnalités ayant été ajoutées et d'autres étant dépréciées.