Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

Automatic Interaction Diagram Generation of Vue.js-based Web Applications

A thesis submitted for the degree of
Master of Science (M.Sc.)

at

Hochschule für Technik und Wirtschaft Berlin
in the degree course
Applied Computer Science (Master)

1. Supervisor: Prof. Dr. Gefei Zhang
2. Supervisor: Prof. Dr.-Ing. Hendrik Gärtner

submitted by: B.Sc. Anton Karakochev
matriculation number: 553324
date of submission: February 8, 2021

Abstract

In this thesis an implementation of an n=gram based language model using the Modified Kneser=Ney smoothing algorithm with the open source Apache Spark large scale data processing engine and the open source document oriented database MongoDB will be presented. This language model will then be used to generate sentences from input sets of keywords.

This thesis aims to further explore the concept of interaction diagrams for scenario testing introduced by [ZZ19]. It is applied to a different framework (Vue.js) ... how scenarios in Gherkin generated and extended lists.

–

automatically generate!!!

–

Contents

Chapter 1

Introduction

Chapter 2

Fundamentals and State of The Art

2.1 State of the Art

TODO

2.1.1 TODO some papers

2.2 Scenario Testing of AngularJS-based Single Page Web Applications

Zhang and Zhao [ZZ19] present a method with the goal of achieving better undestanding of AngularJS-based single page applications (SPAs) and also devised a way to specify test coverage criteria based on it. At the center of the proposed method are interaction diagrams, which are used to model the overall data and control flow of an application.[ZZ19]

2.2.1 Abstract Syntax

Zhang and Zhao [ZZ19] model a AngularJS-based SPA as a tuple (T, C, D, E), where

- T is a HTML template, consisting of a set of HTML tags (widgets) ($T = \{h\}$)

- C is a controller (view-model), written in JavaScript. It is modeled as a tuple (V, F, $\$scope$), where F and V are top level variables and functions respectively and $\$scope \in V$ is a distinguished element of V. Further $V(\$scope)$ and $F(\$scope)$ denote all variables and functions of $\$scope$ respectively. $W = V \setminus \{\$scope\}$ denotes top level variables not in scope. Additionally $init \in F$ is defined as an initialization function

- D is a set of data bindings between HTML tags and variable properties of $\$scope$ $D \subseteq \{(h, V(\$scope) \cup F(\$scope))\}$. Given $d = (n, o)$ $source(d) = n$ and $target(d) = o$. For Two-way bindings $D' \subseteq D$ and $\forall d \in D'$ $target(d) \in V(\$scope)$.

- E a set of event handler bindings between HTML tags and function properties of $\$scope$: $E \subset \{(h, F(\$scope))\}$. In addition, for each function $f \in F \cup F(\$scope)$, $R(f) \subseteq V \cup V(\$scope)$ and $W(f) \subseteq V \cup V(\$scope)$ are defined as the values that the given function reads from and writes to. $Inv(f) \in F$ are defined as the functions invoked by f. [ZZ19]

2.2.2 Interaction Diagrams

Zhang and Zhao [ZZ19] define interaction diagrams as a directed graph (N, E) where the set of nodes N is defined as the union of N_H (HTML tag nodes), $N_{\$scope}$ (TODO name), N_js (TODO name). $N_H = \{n_h | (h, v) \in D\}$

$N_{\$scope} = \{n_v | (h, v) \in D\} \cup \{n_e | (h, e) \in E\}$

$N_js = \{n_v | v \in W\} \cup \{n_f | f \in F\}$

n_init is distinguished by an incoming arrow without a starting vertex

Edges:

bindings

$e_d = (target(d), source(d))$ $E_{data} = \{e_d | d \in D\}$

addiionaly if $d \in D'$ also create $e'_t = (source(t), target(t))$ and $E'_data = \{e_d | d \in D'\}$

for events (E_{event}) $(h, f) \in E$,

write E_W (f, v) where $f \in F \cup F(\$scope), v \in W(f)$

read E_R (v, f) where $f \in F \cup F(\$scope), v \in R(f)$

invoked E_{Inv} (f, v) where $f \in F \cup F(\$scope), v \in I(f)$

E_{init} default values of widgets for each $h \in T$ where $\nexists v | (h, v) \in D$ create an edge (h, v) explained in a lot of detail in [ZZ19, p. 9]

2.2.3 Testing and Interactions

Zhang and Zhao [ZZ19] define an interaction as a round of user input including updates to the widgets by the application. Interaction can be triggered explicitly by the user (by invoking an event handler) or implicitly while the user is updating data. [ZZ19]

Given the interaction diagrams as described in 2.2.2 it is possible to derive which widgets get updated by a user input action or set up by the initial function. Zhang and Zhao [ZZ19] define it formally as follows:

> Given a node $n \in N_H \cup \{init\}$, we say a node $m \in N_H$ *reacts* to n iff
>
> 1. $\exists n_0, n_1, n_2, \ldots, n_k \in N, n_0 = n, n_k = m$ such that for each $0 \leq i < k$ $(n_i, n_{i+1}) \in E$, and
> 2. $\forall n_p, 1 < p \leq k$ and $\forall e \in E, target(e) = n_p$ it holds that $e \notin E_{event}$
>
> We write $l(n)$ for the set of all nodes representing the widgets that react to n. This set contains the widgets that are automatically updated upon user input, and thus constitute an interaction.

For example, in order for the widget n, which was clicked by the user, to update the widget m, m must be reachable from n by following the directed edges of the interaction diagram and only the first edge can be an event-handling edge.

What is crucial is that the interactions $l(n)$ define an upper bound of what can be updated, i.e. what might get updated. Nevertheless, this information is sufficient in order to be able to define coverage criteria [ZZ19].

2.2.4 Coverage Criteria

Interactions should not be tested in isolation and in order for tests to make sence, interactions as preconditions are required [ZZ19]. In order to define coverage criteria, Zhang and Zhao [ZZ19] extend their notation, as described in 2.2.2, by defining - $\mathcal{I} = \{w \in T | l(w) \neq \emptyset\}$ all widgets, that result updates.

A sequence of user interactions, including the initial function is refered to as a *scenario* $A = (a_0, a_1, \ldots, a_n)$ where $a_0 = init$ and $\forall 0 < k \leq n, a_k \in \mathcal{I}$. The widgets, to which a scenario reacts, are equal to the widgets to which the last widget in the scenario reacts - $l(A) = l(a_n)$.

The set of scenarios is generated by starting with the initial scenario, containing only the initial function $S_0 = \{(init)\}$ and prolonging it iteratively by each widget, where the user can take an action. This is terminated once all $i \in \mathcal{I}$ are included in at least one scenario. Formally: For $n > 0, S_{n+1} = \{p \in S_n, x \in l(p) \cap \mathcal{I}\}$

Based on the scenario sets Zhang and Zhao [ZZ19] define the following coverage criteria:

- Each set S_n of test scenarios should be tested.
- For each given S_n, each $p \in S_n$ should be tested.
- For each given p, each $w \in l(p)$ should be tested. That is, there should be a test case for each widget that may be modified after the scenario p.

.. example, which will be implemented here in vue and described in sec xyz.

2.3 Scenario Testing

Scenario testing, was originally introduced in Kaner [Kan03] and later as Kaner [Kan13]. The author defines scenarios as hypothetical stories, which aid a person in understanding a complex system or problem. Scenario tests are tests, which are based on such scenarios. [Kan13, p. 1] Further, [Kan03, pp. 2–5] defines five characteristics, which make up a good scenario test as follows: A Scenario test must be

- based on a story - based on a description of how the program is being used
- motivating - stakeholders have interest in this test succeeding and would see to it's resolution
- credible - probable to happen in the real world

- complex - complex use, data or environment

- easy to evaluate - it should be easy to tell if the test succeded or failed based on the results

Kaner [Kan13] describes the biggest advantages of scenario testing to be - understanding and learning the product in early stages of development(1), connecting of testing and requirement documentations(2), exposing shortcomings in delivering of desired benefits(3), exploration of expert use of the program(4), expose requirement related issues(5).

2.4 Behavior-Driven Development

Behavior-Driven Development (BDD), pioneered by North [Nor06] is a software development process , that combines principles from Test-Driven Development and Domain-Driven design [EE04].

Its main goal is to specify a system in terms of its functionality (i.e. it's behaviors) with a simple domain-specific language (DSL) making use of English-like sentences. This stimulates collaboration between developers and non-technical stakeholders and further results in a closer connection between acceptance criteria for a given function and matching tests used for its validation.

BDD splits a user story into multiple scenarios, each formulated in the form of *Given*, *When*, *Then* statements, respectively specifying the prerequisite/context, event and outcomes of a scenario.

[TODO] example here

At present ... there based on the division of behavior descriptions and behaviors. Such as Jest/Jasmine combine behavior descriptions and behaviors into one, whereas as Cucumber uses a DSL named Gherkin to specify the behavior descriptions and provides a set of tools to generate behaviors.

2.4.1 the other one(forgot name)

splitting into ...

2.4.2 Gherkin Language

2.5 Model-View-ViewModel

Model-View-ViewModel (MVVM) is a design pattern, which helps in creating a clear separation between business and presentation logic and User interface (UI) of an application. [Bri17, pp. 7–9]

In MVVM there are three core components - the view, model and view model. Those components are clearly separated from each other - the view is aware of the view model and the view model is aware of the model. However, this does not hold in reverse - the model is unaware of the view model and the view model is unaware of the view.

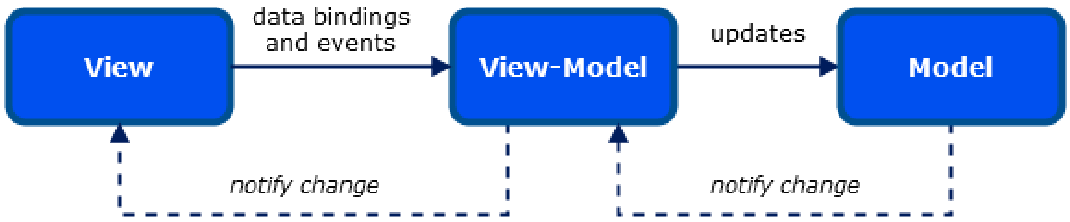

Figure 2.1: MVVM design pattern overview, adapted from [Bri17, p. 7]

2.5.1 View

The view is what the user sees. It is responsible for the structure, layout and appearance of the application.

2.5.2 View-Model

The view model implements event handlers and properties, to which the view can bind to. It also notifies the view of any changes to the underlying data. It defines the functionality, offered by the UI, but the view determines how it is presented.

2.5.3 Model

The model encapsulates the data of the application and validation its logic.

2.6 Vue.js

Vue.js [vue21a] is a progressive front end framework for building user intefaces and single-page applications based on the MVVM design pattern described in 2.5 [Mac18] [21a].

2.6.1 Components

At the core of Vue.js are components, which are small, self-contained, composable and often reusable custom elements. Almost any type of application can be represented as a tree of components [21a].

In more concrete terms, a Vue.js component is a single file with the extension of *.vue*, which consists of a *template*, *script* and optional *style* part. The *template* is a HTML-based template, which can be parsed specification compliant browsers and HTML parsers. It can contain other components or html elements and is equivalent to the *view* in MVVM.

The *code* section of a Vue.js component includes the view-model of the component. It has a special json object *data*, which is equivalent to the MVVM model. The *script* part of a computed includes css-like styles.

data binding is a general technique that binds data sources from the provider and consumer together and synchronizes them.

2.6.2 Reactivity

...enables data binding

2.6.3 Directives

Vue.js enables one way bindings(from source - data to target - component or html tag) via the *v−bind* (line X,Y) or *moustache* syntax (line Z). Bindings can contain expressions (line X,Y).

Two way binding can be achieved using the *v-model* directive (line,X,Y,Z). Event handlers can be bound by the method name or also expressions.

2.6.4 Data Binding

Vue.js provides support for various forms of Data Binding via a special syntax. Both the data and computed objects of a *Vue.js* component are reactive

Via a special syntax *Vue.js* - one way - two way - event bindings - inline expressions - computed properties

2.6.5 Vue.js directives

2.6.6 Structure of a Component

(template, code etc.)

bindings two way, one way

g - data - computed properties

template part code part bindings

2.7 ESLint

ESLint [21b] is a linting tool (linter) for ECMAScript/JavaScript. Linters are static code analysis tool, which can be used to flag and potentially automatically fix common code issues and enforce consistent code styling.

2.7.1 Architecture

At a very high level, ESLint consists of

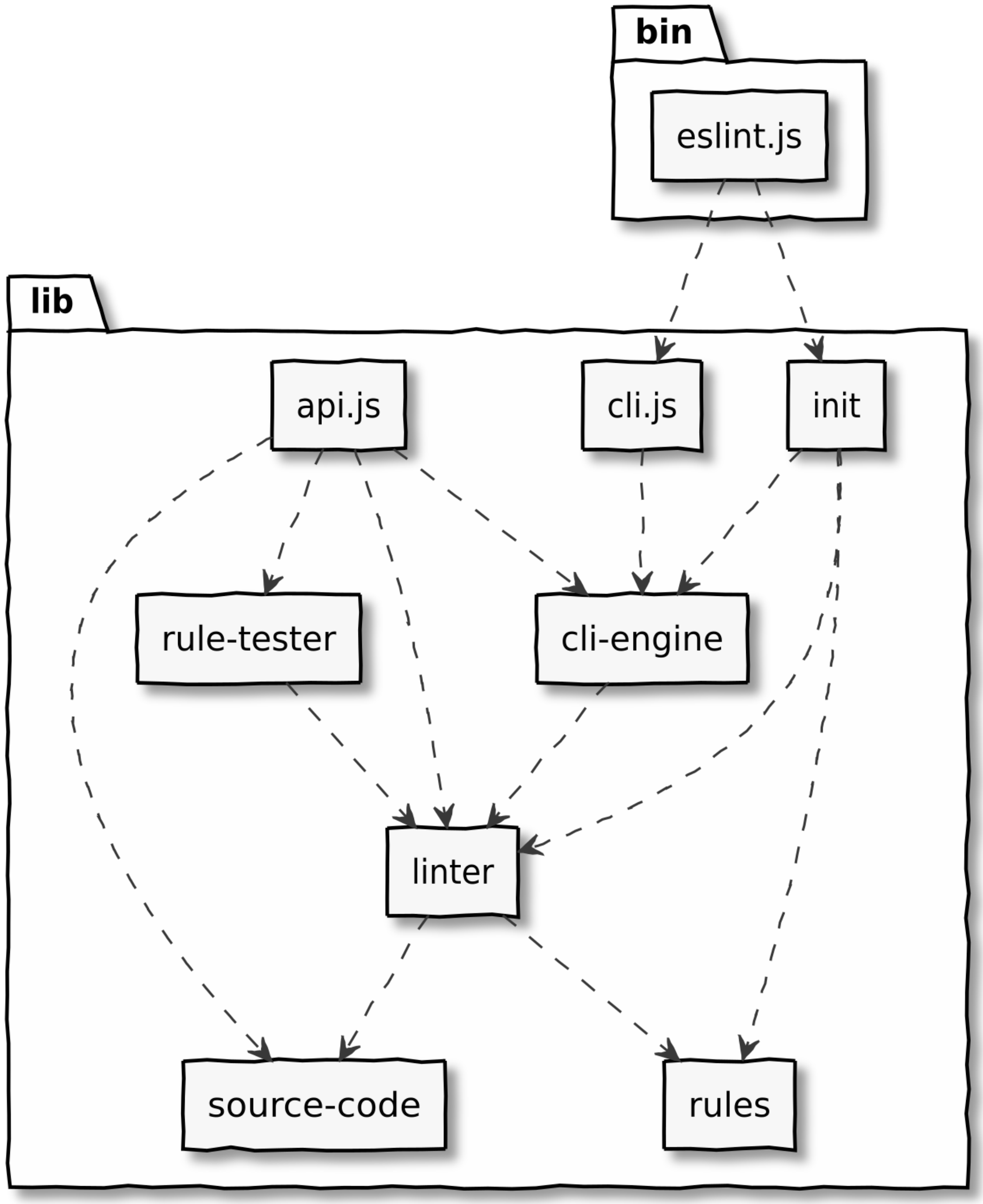

Figure 2.2: ESLint Architecture taken from [21c]

2.8 Rules

At the core of ESLint are rules. Rules are extensible pieces of code, bundled as plugins, which can be used to verify various aspects of code. An example would be a rule, which checks for matching closing paranthesis. Each rule consists of a *metadata* object and a *create* function. The metadata object includes metadata such as documentation strings, the type of the rule and whehter it is fixable or not.

Based on type, rules can be either *suggestions*, *problems* or *layout*. Suggestions indicate some improvement, but are not required and would not cause the linting to fail. Problems on the other hand would result in a linting failure. Layouts are rules that care mainly about the formatting of code, such as whitespaces, semicolons, etc.

If the fixable property is specified, it indicates that the errors reported by this rule can be automatically fixed. This can be applied via the *−−fix* command line option. It has two possible values - *code* or *whitespace* indicating the type of fixes, that this rule would apply. For example in Integrated development environments (IDEs) fixable *code* errors would show a fix shortcut displayed next to them and *whitespace* rules could be applied when saving the file.

The *create* function of rules takes as arguments a *context* and returns an object of methods which are called by ESLint for each node based on the Visitor pattern while traversing the Abstract syntax tree (AST). ESLint provides a very powerful matching mechanism for specifying what nodes to match called selectors [21d] inspired by estools [est21a].

TODO custom architecture image

TODO what are selectors

TODO what does context provide?

2.8.1 AST Explorer

An incredibly useful tool when working with ASTs is AST Explorer, developed by Kling [Kli21]. It enables the exploration of syntax tree generated by various parsers and also includes the vue-eslint-parser [vue21b]

2.8.2 ESTree AST

By default ESLint uses the [esl21] parser to parse JavaScript source code into an AST as defined by ESTree specification [est21b]. When Parsing *.vue* files ESLint uses this parser for the code inside the *<script>* tag.

2.8.3 ESLint Parser Vue AST

In order to parse the *<template>* section of *.vue* files, ESLint uses the vue-eslint-parser [vue21b]. This parser outputs an AST compliant with their own ASTspecification, defined in [vue21c].

Chapter 3

Concept

3.1 Parsing Vue.js

3.1.1 Assumptions

It is assumed that the Vue.js code, for which interaction diagrams are going to be generated, compiles and does not contain syntactical errors. No checks are performed in order to verify that. Naturally, logical errors are not an issue.

3.1.2 Limitations

In order to be able to generate interaction diagrams, which capture every aspect of Vue.js, the generation must be directly based on an AST, which covers every possible syntax, such as [vue21b].

This thesis only includes the following features of Vue.js:

- Event handlers (including anonymous method syntax and method reference syntax)
- Any one or two-way binding expressions (*v−model*, *v−bind*, "moustache", *v−if*) excluding *v−else*
- nested *v−for* statements for lists, excluding iterating through properties of an object or iteration with index (property zipped with index)
- distinguishing between properties and computed properties
- complex object and nested lists models
- methods, including the resolution of arguments, they have been called with (excluding methods called with other methods as arguments)

3.1.3 AST

```
grammar js_simplified;

program: bindings methodDefinitions createdMethod topLevelProperties
    computedProperties;

topLevelProperties: thisIdentifier*;
methodDefinitions: methodDefinition*;
createdMethod: methodDefinition;
computedProperties: (thisIdentifier reads writes calls)*;

methodDefinition: thisIdentifier methodArgs reads writes calls;

methodArgs: NAME_IDENTIFIER*;
reads: accesedVariable*;
writes: accesedVariable*;
```

```
calls: calledMethod*;

calledMethod: calledMethodIdentifier '(' calledArgs ')';
accesedVariable: identifier;
calledArgs: (calledMethod | accesedVariable)*;

bindings: binding*;
binding: tag accesedVariable | calledMethod BINDING_TYPE;

tag: name tagId loc;
tagId: LINE '_' COLUMN '_' LINE '_' COLUMN;
name: UNICODE | identifier;
loc: start end;
start: LINE COLUMN;
end: LINE COLUMN;

calledMethodIdentifier: THIS id* NAME_IDENTIFIER | id* NAME_IDENTIFIER;

thisIdentifier: THIS identifier;
identifier: NAME_IDENTIFIER id*;

id: NUMERIC_INDEX | GENERIC_INDEX | NAME_IDENTIFIER;

//terminals, tokens
LINE: [0-9]+;
COLUMN: [0-9]+;

BINDING_TYPE: 'event' | 'one-way' | 'two-way';
GENERIC_INDEX: 'i' | 'j' | 'k' | 'l' | 'm' | 'n';
THIS: 'this';

NUMERIC_INDEX: [0-9];
NAME_IDENTIFIER:  JS_IDENTIFIER;
JS_IDENTIFIER:  (UNICODE | '$' | '_') (UNICODE | '$' | '_' | [0-9])*;
UNICODE: [\u0000-\uFFFF];
```

A Vue.js SPA, including all the necessary information for 3.1.2, can be defined using the above grammar.

The application consists of *bindings methodDefinitions* a *createdMethod*, *topLevelProperties* and *computedProperties*.

The *topLevelProperties* represent the *data* object of the Vue.js *script* tag. Each property will be represented flattened, as a list of identifiers and prefixed with *this*,in order to indicate it belongs to the top level data object. For example *problem:{a:0, b:0}* will be represented as follows:

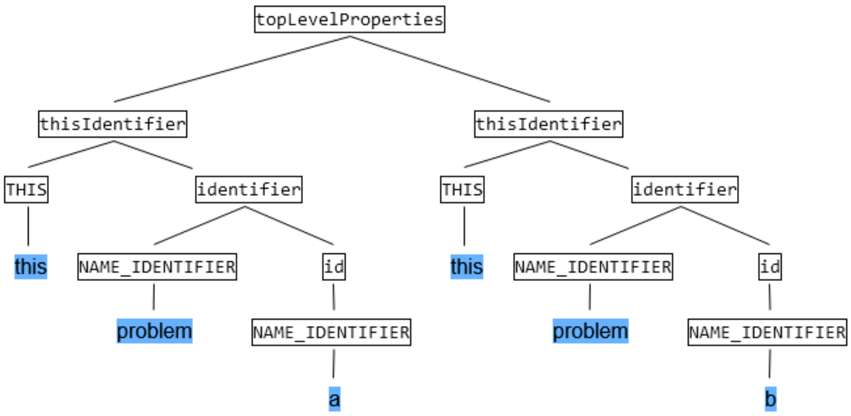

Figure 3.1: AST for top-level example

Bindings can be obtained from the Vue.js *template*.

Each binding consists of an HTML tag, an a list of binding sources for that tag - pairs of variable or method call and a binding type. The binding type represents the type of the binding - either event, one-way or two-way. Two-way bindings are only valid with properties, whereas for events and one-way bindings, both method calls and properties are possible, since in Vue.js a binding source could be an expressions defined as an inline anonymous functions (*<div v−if="value === true"/>*). The binding sources are a list, since a tag could have multiple different bound properties, or a bound expression. The information about how exactly the properties are bound, if it is the same type of binding, is discarded.

Method calls include the parameters they have been called with - other methods or just variables. It is also possible to call methods with binary expressions - those are represented as a special method, which takes 2 parameters - the left and right side operators of the binary expression. Expressions with multiple terms can be represented as multiple binary expressions. This representation loses information such as the order of operations, but since we are only interested in which properties are being accessed, this loss does not pose an issue.

A special case is accessing lists. For example *<div v−bind="problems[0].a"/>* would result in the following:

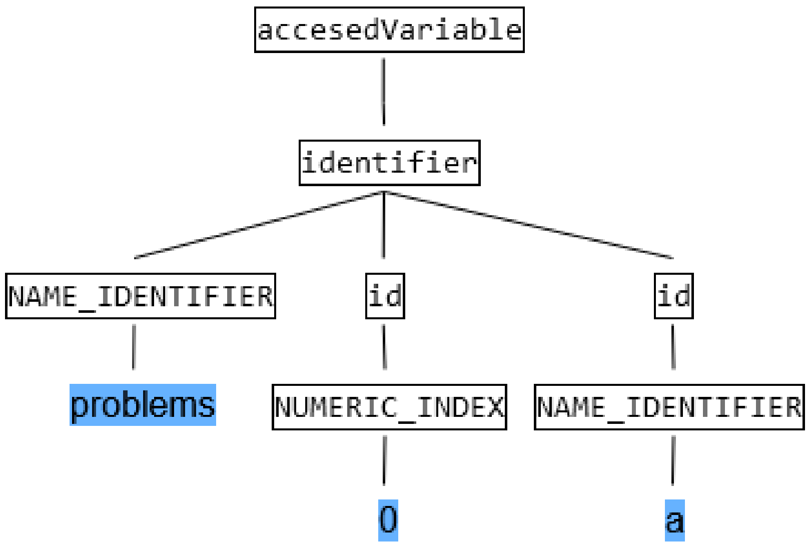

Figure 3.2: AST for example

v–for statements, are substituted:

```
<ul>
  <li v-for="subject in subjects" :key="subject.id">
    {{ subject.problems[0] }}
  </li>
</ul>
```

results in

```
subjects[i].problems[0]
```

which in term produces the following AST:

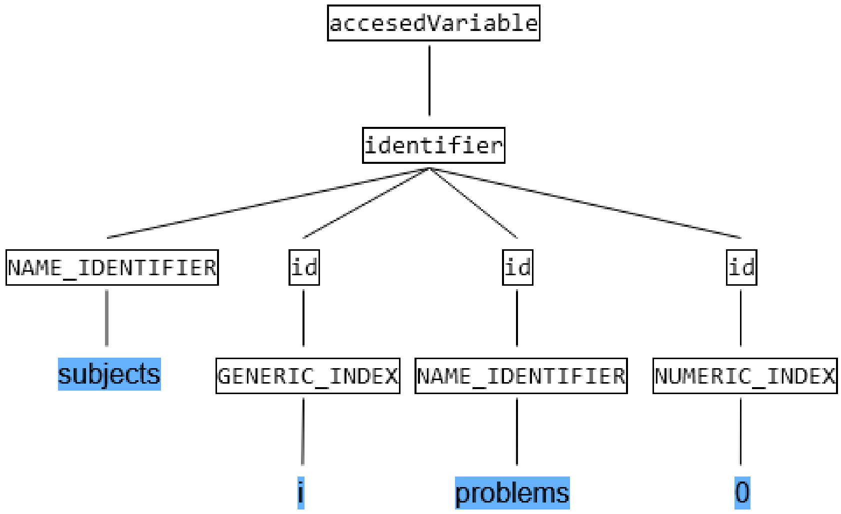

Figure 3.3: AST for example

Nested lists are also possible and would result in multiple generic indices being added.

Each tag includes its location in the source code (starting and ending line and column), which can also be used as an identifier. Tags also have humanly readable names, which are either equal to the the text of the tag, if it exists, or to the identifier of the first binding.

methodDefinitions include all methods definitions from the *method* object of the *view−model*. Each method definition consists of the following

- identifier - an identifier, equal to *this* followed by the name of the method
- arguments - names of arguments, each of which is a simple name identifiers
- reads - variable it reads from
- writes - varaibles it writes to
- calls - method calls, including arguments, same as for bindings

computedProperties are similar to *methodDefinitions* with the exception that they do not have arguments. Albeit bad practise, it is still possible for computed properties to have side effects and therefore it was decided to model them as methods.

3.2 Interaction Diagram Generation

Interaction diagrams can be generated from the simplfied Vue.js AST in the following way. For each binding (tag, binding sources tuple) in bindings $(t, S) \in bindings$:

for each binding source $s \in S$ if it is an accesedVariable v check if it is a computed property by looking up the *computedProperties*. If true, treat it as a method, which will be explained below. If false, check if it is a property, by checking if there is a top level property in $topLevelPropertiers$, which starts with the same identifier (ignoring the *this* prefix). If true, prefix *this* and create a node for each *id* in *identifier*. Create a Globally Unique Identifier (GUID) for each node, by starting from *this* and in sequence iterating over each *id* in *identifier* concatenate its *id* with the *id* of the node before it and also set it as its parent. Connect those nodes with unidrectional edges starting from *this* and label each edge *data*. Assign a label to each node, equal to its *id* if it is a *NAME_Identifier*, or else to the name of the previous node including the name of the current one in square brackets.

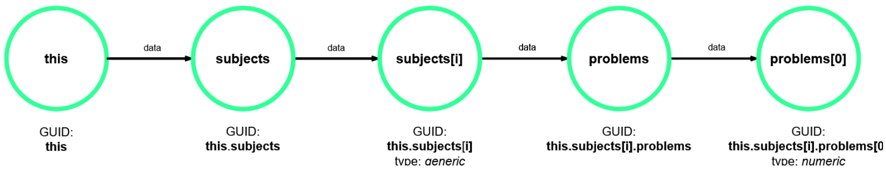

Figure 3.4: Nodes created for identifiers of 3.3

If the binding source s is a *calledMethod*, check if it exists in *methodDefinitions* by excluding the *this* prefix. If true, prefix it with *this* and create a node for it. Replace each of it's arguments, which cannot be found in *topLevelProperties* or *computedProperties* by a fixed word such als *OTHER* or $*$, join them with , and surround them with brackets. Concatenate it with the name of the method to form the GUID of the method. Remove *this* identifiers from the method name and arguments and and set it as its label.

The node for the method itself is now completed, the next step is to create nodes for it's the properties it *reads* and *writes* and for other methods it *calls*, if they do not yet exist based on its *methodDefinition*. The first step is to substituted the method definition arugments with the called arguments and update all *reads*, *writes* and *calls* referencing them. Now *reads*, *writes* and *calls* which do not start with *this* can be removed since they are not properties. After this is done, create a node for each property in *reads* and *writes* in the same way as described above. For each property in *writes* add an edge with the property as the source and the current method as the sink. For each property in *calls* add an edge with current method as the source as the property as the sink.

For each method in *calls*, which does not yet exist in the graph, recursively repeat the above process and add and edge from the current method to it labelled *calls*.

If any nodes were created in the previos steps, create a node for the tag t with a GUID *tagId* and label *name*. Based on the binding type create edges to connect t and each of the nodes. If the binding type is an event binding, create an edge with source t and sink *node* and label it *event*. If the binding type is one-way, create an edge with source *node* and sink t. If the binding type is two-way, create an edge

with source t and sink *node* and label it *event* and a second edge with source *node* and sink t. In essence two-way bindings create the same edges as a combination between one-way and event bindings.

For the initial method - $createdMethod$ create a node with GUID and name equal to *created*. Resolve its $reads$, $writes$ and $calls$ in the same maner as methods, described in the previos section.

Due to dynamic nature...

3.3 Scenario Generation

In order to generated scenarios in Gherkin, interactions can be sliced in a similar manner as described by Zhang and Zhao [ZZ19] and summarized in 2.2.2.

Let N denote the set of all nodes in the graph and E denote all edges in the graph. Let $n \in N$, $m \in N$ be any two nodes in the graph and let $(n, m) \in E$ represent an edge from n to m. Let $type(n)$ be a function, that returns the type of a node and $label(n, m)$ be a function that returns the label of the edge from n to m. Let $E_{out}(n)$ be a function, which returns all outgoing edges of n. Let $E_{in}(n)$ be a function, which returns all incoming edges to n.

Let N_I denote all nodes, that the user can interact with. A node $n \in N$ is also in N_I if $\exists e \in E_{out}(n)$ where $label(e) = event$. Let N_H denone all html tag nodes. a node $n \in N$ is also in N_H if $type(n) = tag$. Given a node $n_H \in N_H \cup created$, a node $m_H \in N_H$ reacts to n iff

1. $\exists n_0, n_1, n_2, \ldots, n_k \in N, n_0 = n_H, n_k = m_H$ such that for each $0 \leq i < k$ $(n_i, n_{i+1}) \in E$, and $label((n_i, n_{i+1})) \neq event$ and if $label(n_i, n_{i+1}) \neq calls$ $\forall n_{i+1_{in}} E_{in}(n_{i+1})$ $label(n_{i+1_{in}}) \neq event$

Analogous to [ZZ19] let $l(n)$ denotes all nodes, that react to n. A sequence of user interactions, starting with the initial function is refered to as a *scenario* $A = (a_0, a_1, \ldots, a_n)$ where $a_0 = created$ and $\forall 0 < k \leq n, a_k \in N_I$.

Define the HTML tags, to which a scenario reacts, to be equal to the tags to which the last tag in the scenario reacts - $l(A) = l(a_n)$.

The set of scenarios is generated by starting with the initial scenario, containing only the initial function $S_0 = \{(created)\}$. It is then prolonged by all tags $n \in N_I$, representing that the user can click anywhere resulting in N_I scenarios of length 2. Scenarios of length 3 are generated by taking the previous ones of length 2 S_2 and for each scenario $SinS_2$ generate up to N_I new scenarios by prolonging S with $n | n \in l(S) \cap N_I$ This is repeated up to k times, where k is a constant set by the user. Then each set is collapsed() and later only unique interaction are generated.

and prolonging it iteratively by each widget, where the user can take an action.

Gherkin scenarios templated can then be obtained by the following tempalte: Scenario - $n_0, n_1 \ldots n_k$ Given - $n_0, n_1 \ldots n_{k-1}$ When - n_k Then - $l(n_k)$

Appendix A

Appendix

List of Figures

List of Tables

Bibliography

[ZZ19] Gefei Zhang and Jianjun Zhao. "Scenario Testing of AngularJS-Based Single Page Web Applications". In: *International Conference on Web Engineering*. Springer. 2019, pp. 91–103.

[Kan03] Cem Kaner. "The Power of'What If... and Nine Ways to Fuel Your Imagination: Cem Kaner on Scenario Testing". In: *Software Testing and Quality Engineering* 5 (2003), pp. 16–22.

[Kan13] Cem Kaner. "An introduction to scenario testing". In: *Florida Institute of Technology, Melbourne* (2013), pp. 1–13.

[Nor06] Dan North. "Behavior modification". In: *Better Software* 8.3 (2006), p. 26. URL: `https://dannorth.net/introducing-bdd/`.

[EE04] Eric J Evans and Eric Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[Bri17] David Britch. *Enterprise Application Patterns using Xamarin. Forms*. 2017. URL: `https://github.com/dotnet-architecture/eBooks/blob/master/current/xamarin-forms/Enterprise-Application-Patterns-using-XamarinForms.pdf`.

[vue21a] vuejs. *vue*. [Online; accessed 29. Jan. 2021]. Jan. 2021. URL: `https://github.com/vuejs/vue`.

[Mac18] Callum Macrae. *Vue. js: Up and Running: Building Accessible and Performant Web Apps*. " O'Reilly Media, Inc.", 2018.

[21a] *Introduction Vue.js*. [Online; accessed 29. Jan. 2021]. Jan. 2021. URL: `https://vuejs.org/v2/guide`.

[21b] *ESLint - Pluggable JavaScript linter*. [Online; accessed 25. Jan. 2021]. Jan. 2021. URL: `https://eslint.org`.

[21c] *Architecture*. [Online; accessed 27. Jan. 2021]. Jan. 2021. URL: `https://eslint.org/docs/developer-guide/architecture`.

[21d] *Selectors*. [Online; accessed 27. Jan. 2021]. Jan. 2021. URL: `https://eslint.org/docs/developer-guide/selectors`.

[est21a] estools. *Esquery*. [Online; accessed 27. Jan. 2021]. Jan. 2021. URL: `https://github.com/estools/esquery`.

[Kli21] Felix Kling. *AST Explorer*. [Online; accessed 27. Jan. 2021]. Jan. 2021. URL: `https://github.com/fkling/astexplorer`.

[vue21b] vuejs. *vue-eslint-parser*. [Online; accessed 27. Jan. 2021]. Jan. 2021. URL: `https://github.com/vuejs/vue-eslint-parser#readme`.

[esl21] eslint. *Espree*. [Online; accessed 25. Jan. 2021]. Jan. 2021. URL: `https://github.com/eslint/espree`.

[est21b] estree. *estree*. [Online; accessed 27. Jan. 2021]. Jan. 2021. URL: `https://github.com/estree/estree`.

[vue21c] vuejs. *vue-eslint-parser*. [Online; accessed 27. Jan. 2021]. Jan. 2021. URL: `https://github.com/vuejs/vue-eslint-parser/blob/master/docs/ast.md`.