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 9, 2021

Abstract

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) ... - automatically generate based on vue js code - lists - objects - computed property - scenarios in Gherkin generated

Contents

1 Introduction **5**

2 Fundamentals and State of The Art **6**

2.1 State of the Art 6
2.2 Scenario Testing of AngularJS-based Single Page Web Applications 6
2.2.1 Abstract Syntax 6
2.2.2 Interaction Diagrams 7
2.2.3 Testing and Interactions 7
2.2.4 Coverage Criteria 8
2.3 Scenario Testing 8
2.4 Behavior-Driven Development 9
2.4.1 Gherkin Language 9
2.5 Model-View-ViewModel 9
2.5.1 View 10
2.5.2 View-Model 10
2.5.3 Model 10
2.6 Vue.js 10
2.6.1 Components 10
2.6.2 Reactivity 11
2.6.3 Directives 11
2.6.4 Data Binding 11
2.6.5 Vue.js directives 11
2.6.6 Structure of a Component 11
2.7 ESLint 11
2.7.1 Architecture 11
2.8 Rules 12
2.8.1 AST Explorer 13
2.8.2 ESTree AST 13
2.8.3 ESLint Parser Vue AST 13
2.8.4 Selectors 13

3 Concept **14**

3.1 Parsing Vue.js 15

3.1.1 Assumptions 15

3.1.2 Limitations 15

3.1.3 AST 15

3.2 Interaction Diagram Generation 19

3.2.1 Variable Identifiers 20

3.2.2 Object representation 21

3.2.3 List representation 21

3.2.4 Method representation 22

3.3 Scenario Generation 25

4 Implementation **26**

4.1 Project Structure and Overview 27

4.1.1 Project structure 27

4.2 Parsing Vue.js 28

4.2.1 Common Data Types 28

4.2.2 Top Level Properties 29

4.2.3 Bindings 30

4.2.4 Method Definitions 31

4.2.5 Output 32

4.3 Interaction Diagram Generation 33

4.4 Scenario Generation 35

4.5 Usage 35

5 Testing and Evaluation **36**

6 Conclusion **38**

A Appendix **40**

Chapter 1

Introduction

1 p motivation, etc

Chapter 2

Fundamentals and State of The Art

2.1 State of the Art

TODO

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).

TODO double check, write as text $N_H = \{n_h | (h, v) \in D\}$

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

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

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: Define $A \oplus x = a_0, a_1, \ldots, a_n, x$ For $n > 0, S_{n+1} = \{p \oplus x | p \in S_n, x \in l(p) \cap \mathcal{I}\}$

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.

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? cut shorter

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 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.

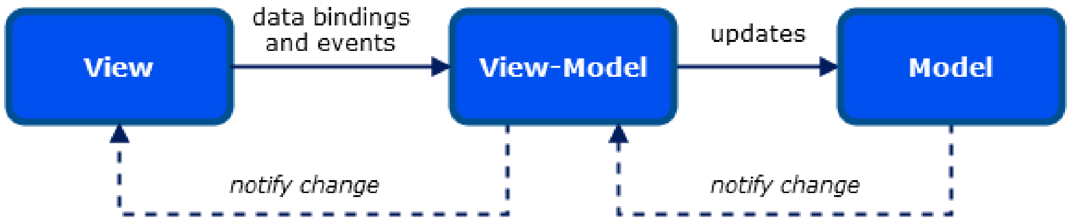

Figure 2.1: MVVM design pattern overview, adapted from [Bri17, p. 7]

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

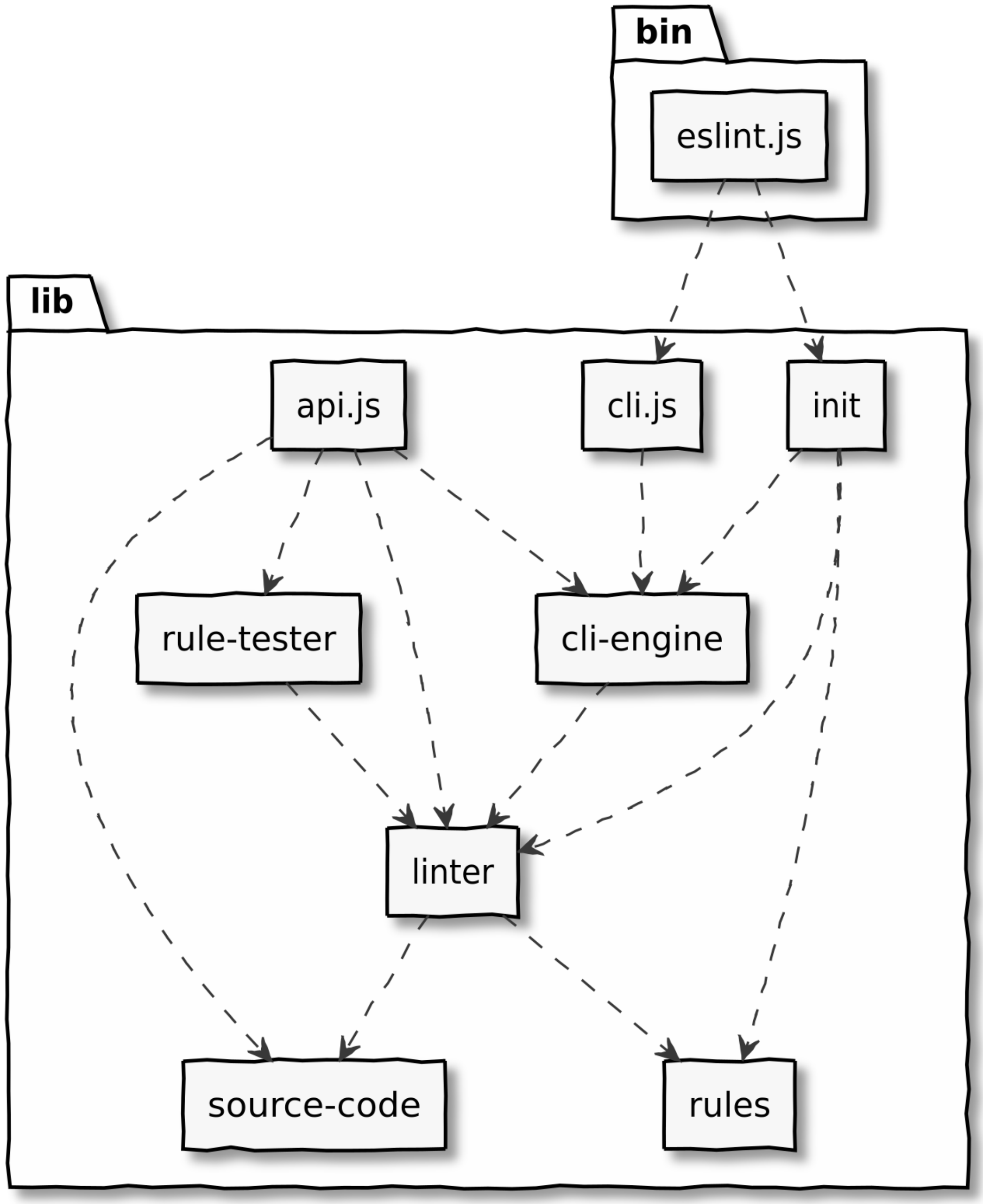

Figure 2.2: ESLint Architecture taken from [21c]

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 cut shorter

TODO custom architecture image

TODO what are selectors

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].

TODO add here from ast def

2.8.4 Selectors

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].

The approach proposed here 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 $v{-}else$
- $v{-}for$ 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 lists (non-nested) 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 vue_simple;

program: bindings methodDefinitions createdMethod topLevelProperties
    computedProperties;

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

methodDefinition: methodDefinitionIdentifier 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 bindingSource+;
bindingSource: (accesedVariable | calledMethod) (EVENT_BINDING |
    ONE_WAY_BINDING)
               | accesedVariable TWO_WAY_BINDING;

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

calledMethodIdentifier: methodDefinitionIdentifier | id* NAME_IDENTIFIER;

methodDefinitionIdentifier: THIS NAME_IDENTIFIER;
thisIdentifier: THIS identifier;
identifier: NAME_IDENTIFIER id*;


id: NUMERIC_INDEX | GENERIC_INDEX | NAME_IDENTIFIER;

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

EVENT_BINDING: 'event';
TWO_WAY_BINDING: 'two-way';
ONE_WAY_BINDING: 'one-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:

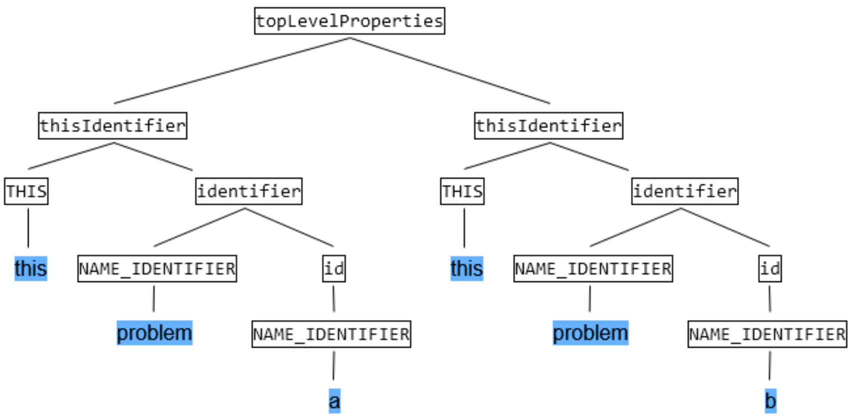

Figure 3.1: AST for top-level example

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:

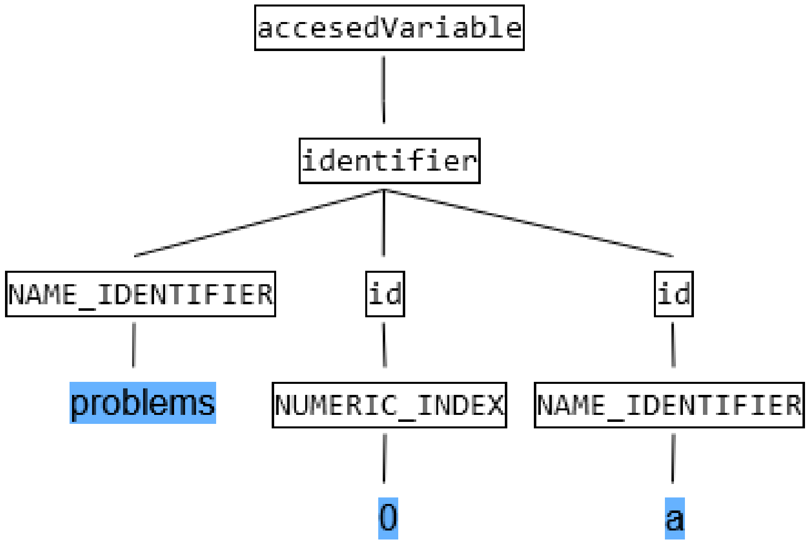

Figure 3.2: AST for example

$v-for$ 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:

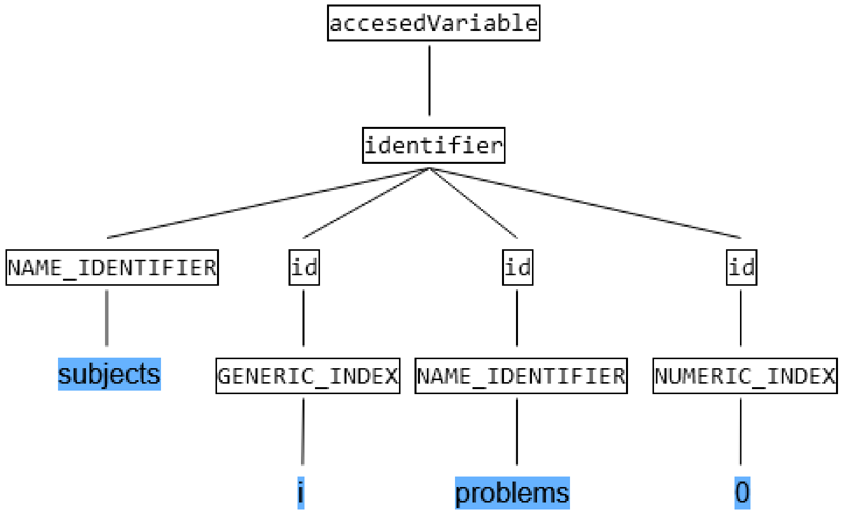

Figure 3.3: AST for example

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 they were modelled as methods.

3.2 Interaction Diagram Generation

The simplified Vue.js AST can be used to create a directed graph, which will represent the interaction diagrams. It is hard to directly generate this graph, therefore the capabilities of a directed, compoundend graph will be leveraged and later on converted to a directed graph.

Vertices in this graph have the following properties

1. Globally Unique Identifier (GUID) - used to reference and globally identify the vertex

2. *label* - the name of the vertex, which is going to be displayed

3. *type* - the type of the vertex (data, tag or method). Additionally for data vertices: numeric, generic or undefined (representing simple data vertices)

4. *loc* - defined only on tag verices. Their location in the source code

5. *parent* - defined only on vertices of type 'data'. A GUID of another vertex, used for a child/parent relationship (compound graph).

Edges in the graph are directed and each have a label property, which is one of 'event', 'calls' or 'simple'.

The core idea of the algorithm is to generate vertices only for nodes which are being accessed instead of the whole application. A second pass of the data is also needed to add additional edges for lists.

3.2.1 Variable Identifiers

Variable identifiers are represented by *identifier* and *thisIdentifier* in the AST 3.1.3. For the *this* and for each *id* node in the *identifier* or *thisIdentifier* a vertex is created in order. Those vertices are connected using unidrection edges, labeled with 'data' and also each vertex (excluding the first one) has its parent set to the previous. There is one exception to this process - When accessed from *write* of a *method*, nodes of type *GENERIC_INDEX* are omitted. The reason behind this will be explained in this section 3.2.3;

Each vertex has a GUID equal to the value of its terminal symbol (*NUMERIC_INDEX*, *GENERIC_INDEX* or *NAME_IDENTIFIER*), concatenated with the value of the previous vertix. The *label* of those vertices are equal to the terminal symbol in case of *NAME_IDENTIFIER* and in case of *GENERIC_INDEX* and *NUMERIC_INDEX*, combined with the *label* of the previous vertex using square braces. Set the type of each vertex to 'data'. Add the type 'numeric' to vertices created from *NUMERIC_INDEX* nodes and 'generic' to vertices created from *GENERIC_INDEX* nodes.

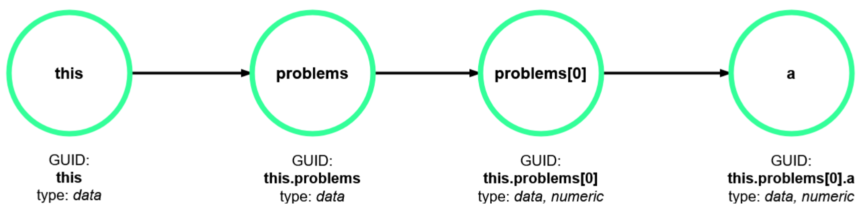

Figure 3.4: Example Graph obtained for the identifier *this.problems[0].a*

3.2.2 Object representation

Using the representation for identifiers in the previous section, objects will result in being displayed dynamically, based on which properties are accessed. Nodes and edges are created on a 'create if non-existent' basis. In the example below, if *this .problem.b* is accessed after *this .problem.a* it will only result in the creation of the node *a* and edge *this .problem* -> *this .problem.b*.

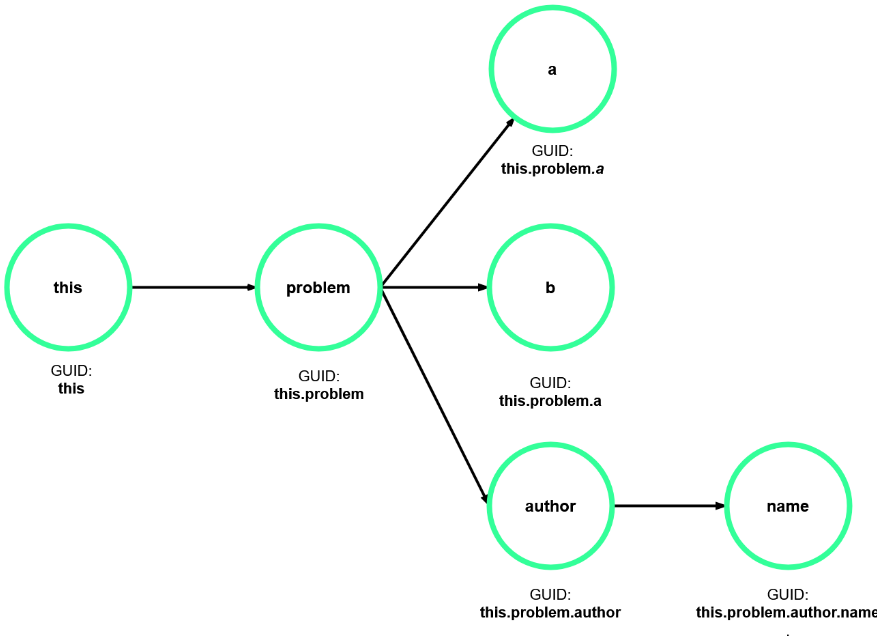

Figure 3.5: Example Graph for the following object accesses - *this .problem.a*, *this .problem.b*, *this .problem.author.name*

Furthermore, updates can be formulated nicely with the above representation. If *problem* were to be changed, it would result in a cascade update of all properties. If *author* would be change, it would only result in a cascading change in *name*.

3.2.3 List representation

Lists will be represented based on the template in 3.6 for a list named P.

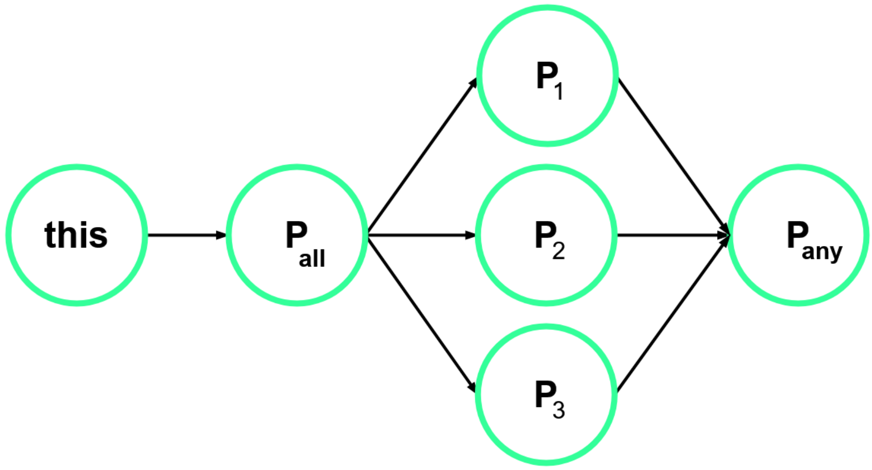

Figure 3.6: Generic List representation

Concrete elements, which are accessed, are detoned as $P_{<index>}$ and additionally a vertex P_{all}, which can be used to update all elements of a list and their properties, is created. Another vertex P_{any} is also created, which can be used to observe once any vertex of P_1, P_2, P_3, $\dots P_n$ changes. If P_1 were to be updated by any method, it would not result in updates to any of P_2, P_3, $\dots P_n$

The same construct can also be leveraged when it comes to properties of list elements. Each top level property of that element will have an *all* vertex, connected to the P_{all} node of the list.

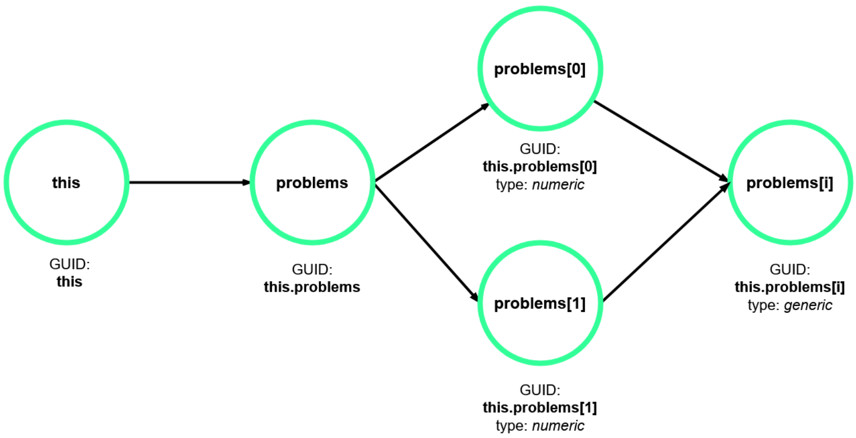

Figure 3.7: Concrete example of a list representation

3.2.4 Method representation

Methods have two related AST nodes - *methodDefinition*, representing the definition of a method and *calledMethod* representing a call of a method.

First it should be determined if a vertex needs to be created for an *calledMethod* node. This is done by looking up based on the name of the *calledMethod* in *methodDefinition*, ignoring *THIS*. If the lookup is succesful a vertex should be created as described below. If not, it should be checked if the method is a method call on a top level property instance. This can be done by comparing if it starts in the same way as one of *topLevelProperties*. If that's the case, it is assumed, that it mutates whole property and the method should instead be treated as a write operation. If both of the above fail, the called method does not belong to context and is of no interest.

The next step is to resolve the names of the arguments it has been called with *calledArgs*. Every argument, that can neither be found in *computedProperties* nor *topLevelProperties* is replace by a fixed word such as $OTHER$ or $*$. In order to obtain the GUID of the vertex, the name of the *methodDefinitionIdentifier* is taken, *THIS* is excluded, and concatened with the the resolved arguments, which are joined with , and surrounded with brackets. The *label* of this node is equal to its name, excluding *THIS* from arguments.

The vertex for the method call is now completed. Multiple calls of this method with the same arguments will all result in the same vertex.

Now vertices for nodes the method interacts with, based on its *methodDefinition*, have to be created. Those include the variables it reads - *reads*, and writes - *writes* and methods it calls - *calls*.

Firstly, the arguments from the *methodDefinition* need to be substituted with the resolved arguments the method was actually called with and update all *reads*, *writes* *calls* referening them. All of them, which do not start with *THIS* can be discarded, since they do not belong to the context. Once filtered out, create a list of vertices for each variable in *reads* and *writes* as described in 3.2.1 and connect the most precise of those (the last of each list) to the method vertex. For the vertices resulting from *writes*, this edge has a label of 'writes' and the property vertex as the source and method vertex as the sink. For the vertices resulting from *reads*, this edge has the method vertex as the source and property vertex as the sink. Finally the process described in this section is repeated recursively for each *calledMethod* node in *calls* and an edge labeled 'calls' is added from the current method vertex to the resulting ones.

Computed property representation

Computed properties are represented similarly to methods, except they cannot have arguments, so no subsitution of arguments is required. When defining their *label* and GUID both are equal to the *methodDefinitionIdentifier*. *reads*, *writes* and *calls* are computed in the same manner as methods.

Combining it all together

Interaction diagrams can be generated from the simplfied Vue.js AST in the following way:

For each *binding* in *bindings*: - for each *tag*, *bindingSource* in *binding*:

Create a vertex for *tag*, with a GUID *tagId* and label *name* and type 'tag'.

if the *binding* is an *accessedVariable*, determine if it is a computed property by looking it up in *computedProperties* and if so, treat it as a computed property, and create vertices as described in 3.2.4. Otherwise determine if it as top level property, by doing a lookup on *topLevelProperties*, treat it as a proeprty and create vertiecs for it as desribed in 3.2.1. In either cases, connect it to the *tag* vertex, based on the binding type. If the *accessedVariable* is neither, it does not belong to context and can be discarded.

if the *binding* is an *calledMethod* create a vertices for it as described in 3.2.4. Connect it to the *tag* vertex, based on the binding type.

Based on binding type, the following edges are created: A) If the binding type is an event binding, create an edge with the tag vertex as a source and the binding vertex as sink and label it 'event'. B) If the binding type is one-way, create an edge with the binding vertex as source and the tag vertex as sink. C) If the binding type is two-way, create both edges - A) and B).

For the initial method - *createdMethod*, create a vertex with GUID and name equal to *created* and create vertices for its *reads*, *writes* and *calls* analogous to methods as described in 3.2.4.

Once all of the above is done, additional edges will need to be added for the *all* vertices of properties of elements inside lists 3.2.3. Also the edges to the *any* vertex will be missing.

'numeric' vertices to the 'generic' one.

This is achievable by first finding all vertices of type 'generic' or 'numeric' and obtaining the parent of each of them. Those parents form a subset of all vertices, that have have 'generic' or 'numeric' vertices as children and adittionaly other properties (properties on list elements, with $type(v) \neq generic, type(v) \neq numeric$). For each of those parent vertices p: Firstly, connect each 'numeric' vertex to the 'generic' one. Recursively connect each child in the tree of the 'numeric' vertex to the child of the tree of the 'generic' vertex with the same name. If either does not exist, no edge is created. Do the same for p and all 'numeric' vertices and the 'generic' vertex.

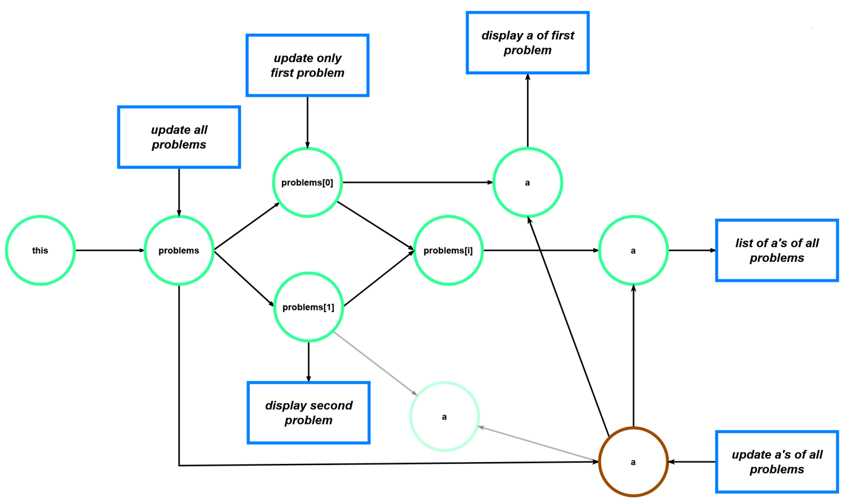

Figure 3.8: Example Graph including list elements with properties

3.3 Scenario Generation

In order to generate 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 $(n, m) \in E$ 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_H 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 $l(A) = l(a_n)$. Define a function that returns the last element in a scenario - $last(A) = a_n$

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. For further steps, only tags, that can be updated are included, so additionaly $n \in l(A)$ must hold. The newly included tag should also not be the same as the last element of the scenario, which means that also $n \neq last(A)$ must hold. Formally: Define $A \oplus x = a_0, a_1, \ldots, a_n, x$. Then

$$S_n = \begin{cases} \{(created)\} & \text{if } n = 0 \\ \{p \oplus x | p \in S_{n-1}, x \in \mathcal{I}\} & \text{if } n = 1 \\ \{p \oplus x | p \in S_{n-1}, x \in l(p) \cap \mathcal{I}, x \neq last(p)\} \end{cases} \tag{3.1}$$

This is repeated up to k times, where k is a constant set by the user.

A 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 - $l(n_k)$

Chapter 4

Implementation

4.1 Project Structure and Overview

The application is written mostly in TypeScript using npm as a package manager and Node.js as a runtime environment.

Notable dependencies are TypeScript, for stricter syntax and types, lodash[21e] for enrichening of collections, graphlib[dag21] for the interaction diagrams graph, babel[21f] as a transcompiler, eslint[21b], estree[est21b] and eslint-plugin-vue[vue21b] for parsing Vue.js code, d3-graphviz[Jac21] in comination with light-server[Che21] for visualization of the generated interaction diagrams. The full list of dependencies can be found in the *package.json* of the project.

4.1.1 Project structure

The main source files of the project and their tests are included in the *src* directory and structured in several packages, each corresponding to a step in the process, except common, which is shared among all steps. Each will be described in more detail in the following sections. Each package includes a models directory, which inludes the data types defined and used in this section. The *web* directory contains code used to view the resulting diagram in the browser. The *scripts* directory includes helper bash scripts, *results* holds snapshots of the results throughout development (with the latest being the current) and *resources* hold various additional files needed.

```
root:.
|-- resources
|   |-- output
|
|-- results
|
|-- scripts
|
|-- src
|   |-- common
|   |   |-- models
|   |
|   |-- main.ts
|   |
|   |-- generator
|   |   |-- models
|   |
|   |-- parsing
|   |   |-- builders
|   |   |-- models
|   |   |-- visitors
```

```
|     |
|     |-- scenarios
|
|-- web
```

4.2 Parsing Vue.js

Instead of implementing a parser, which directly outputs the simplfied Vue.js AST described in 3.1.3 the capabilities of *ESLint - Pluggable JavaScript linter* [21b] and [vue21b] were used. The source files, which handle the parsing reside in the *parsing* directory. The *ESLinter* class provides a wrapper around the Node.js API of ESlint. Custom visitors are implemented in order to extract the necessary nodes from the AST of ESLint. Each visitor has a matching file in *models*, which holds the models specific to that visitor, and a *builder*, which keeps track of the visited nodes and builds the result data type. There are a total of three visitors - one for top level properties, another for bindings and the last one for method defintions, computed properties and the created method.

4.2.1 Common Data Types

Below are the common data types used by all visitors.

```
export type Identifier = This | NameIdentifier | NumericIndex
| GenericIndex;

interface BaseIdentifier {
  readonly name: string;
}
export interface This extends BaseIdentifier {
  name: "this";
  discriminator: IdentifierType.THIS;
}
export interface NumericIndex extends BaseIdentifier {
  discriminator: IdentifierType.NUMERIC_INDEX;
}
export interface GenericIndex extends BaseIdentifier {
  discriminator: IdentifierType.GENERIC_INDEX;
}
export interface NameIdentifier extends BaseIdentifier {
  discriminator: IdentifierType.NAME_IDENTIFIER;
}
```

```
export type Entity = Method | Property;

export interface Property {
  id: Identifiers;
  discriminator: EntityType.PROPERTY;
}
export interface Method {
  id: Identifiers;
  args: ReadonlyArray<Entity>;
  discriminator: EntityType.METHOD;
}
```

Discriminators are used to be able to diferentiate between the types using type guards. The enums themselves are omitted here (*IdentifierType*, *EntityType*). The definitions here are not exactly the same as in the AST 3.1.3 - some constraints are omitted, such as method names having to end on a a *NameIdentifier*. This will be given, since the parsed code would be invalid javascript otherwise.

4.2.2 Top Level Properties

The result of the top level properties has the following data type.

```
export type TopLevelProperties = Array<Property>;

export interface TopLevelPropertiesResult {
  topLevel: TopLevelProperties;
}
```

The top level properties visitor is the simplest of all since it only reacts to the top level *data* node inside the *script* object of the Vue.js SPA, which is a *ObjectExpression* 2.8.3. It can be selected via the following selector:

```

"ExportDefaultDeclaration > ObjectExpression >
Property[key.name = data] ReturnStatement > ObjectExpression"(node){
  ...
}
```

In natural language the selector reads: "Select ObjectExpression nodes, which have a direct parent ReturnStatement, that has an indirect parent Property with a property *key.name* equal to *data* and a direct parent ObjectExpression with a direct parent ExportDefaultDeclaration.

For each of the properties 2.8.3 of the *ObjectExpression* the name of the key (identifier) is stored. If the property is an object (value of *ObjectExpression* 2.8.3) it is concatenated with the previously obtained key. Finally all obtained properties are prefixed with 'this'.

4.2.3 Bindings

The result of the bindings visitor has the following data type.

```
export enum BindingType {
  EVENT = "event", ONE_WAY = "one-way", TWO_WAY = "two-way",
}

export interface Tag {
  id: string;
  loc: Location;
  name: string;
  position?: string;
}

export interface BindingValue {
  item: Entity;
  bindingType: BindingType;
}

export type Binding = { tag: Tag; values: BindingValue[] };
export interface BindingsResult {
  bindings: Binding[];
}
```

The ESlint AST nodes, which are interesting when parsing the bindings are *VElement* 2.8.3, *Identifier* 2.8.3, *MemberExpression* 2.8.3 and *CallExpression* 2.8.3.

A *identifier* 3.1.3, abstracted in /]common/identifier.ts can be a single *Identifier* or a *MemberExpression*, which can contain other *Identifier* nodes or *MemberExpression* nodes. Property Identifiers are extracted by finding the root *MemberExpression* or *Identifier* and traversing it. It is easy to determine if a *MemberExpression* or *Identifier* is the root - its parent is anything but a *MemberExpression*.

A *CallExpression* contains information about the name of the method and the arguments it has been called with, both in the form of nested *MemberExpression* and *CallExpression* nodes. Once again, only the root *CallExpression* node is extracted and converted to a *calledMethod* 3.1.3, abstracted in the *Method* interface in *shared.ts*.

VElement nodes represent any HTML tag, matching a *tag* 3.1.3 abstracted in the *Tag* interface in codetemplate-bindings.ts and contain information about the location of the tag and potentially a *VText* 2.8.3 node, which will be set as its name if it is present. If not present, the name of the tag is equal to the name of the first binding. Therefore, information about tags is extracted once a *VELement* is exited, since all bindings will be known at this point.

Further, the binding type has to be determined. This can be extracted based on the *VAttribute* 2.8.3. Event bindings have a *VAttribute* with a *key.name.name* equal to 'on', two-way bindings equal to 'model' and everything else is interpreted as one-way bindings. This includes moustache statements, *v−bind*, *v−if* bindings and all other except *v−for* statements. This filter be achieved via the powerful *:not* in combination with *:matches* selectors:

```
:not(:matches(
  VAttribute[key.name.name=on],
  VAttribute[key.name.name=model],
  VAttribute[key.argument.name=key],
  VAttribute[key.name.name=for]))
```

With all the above, for example to match all two-way bindings and pass them on to the builder can be done via

```

"VAttribute[key.name.name=model] > VExpressionContainer
:matches(MemberExpression, Identifier, CallExpression)"(
  node) {
  if (utils.isRootNameOrCallExpression(node) &&
  utils.notArgument(node))
    builder.identifierOrExpressionNew(node, BindingType.TWO_WAY);
},
```

Bindings also need to substitute *v−for* statements. This is done by substituting the left side of the *v−for* statement with its right side and a generic index in all bindings that use it.

4.2.4 Method Definitions

```
export interface MethodDefinition {
  id: Identifiers;
  args: ReadonlyArray<Property>;
  reads: ReadonlyArray<Property>;
  writes: ReadonlyArray<Property>;
  calls: ReadonlyArray<Method>;
}
export type MethodDefinitions = Array<MethodDefinition>;

export interface MethodsResult {
  init?: MethodDefinition;
  computed: MethodDefinitions;
  methods: MethodDefinitions;
}
```

All method definition like structures (computed properties, created) and methods are parsed by the visitor defined in *methods.js*. Analogous to how the top level *data* object is selected 4.2.2, the *methods*, *created* and *computed* objects can be selected. The name of the method including its arguments can be extracted from by *Property[value.type=FunctionExpression]* nodes. Using this information, one can have three selectors, one of each type, to determine what is being defined. For example for regular methods:

```

"ExportDefaultDeclaration > ObjectExpression > Property
[key.name = methods] Property[value.type=FunctionExpression]"(node) {
  builder.newMethod(node, MethodType.METHOD);
},
```

Further the properties read, written and methods called need to be extracted. Methods called can be obtained by selecting *CallExpression* nodes. Properties written to can be obtained from the left side of a *AssignmentExpression* 2.8.3. There does not seem be an easy way to select all properties read from. Therefore all accesed identifiers are first stored and everything except reads, that can have an identifier (object properties, variable declarations, variables written to and names of called methods) is subtracted from it, in order to obtain the variables that the method reads from.

The following code can be used to obtain all variables written to by the current method-like in scope.

```
"ExportDefaultDeclaration > ObjectExpression >
:matches(Property[key.name = methods], Property[key.name = created],
Property[key.name = computed]) AssignmentExpression"(node) {
  builder.identifierOrExpressionNew(node.left, AccessType.WRITES);
},
```

4.2.5 Output

Combinging all of the above, the following data structure is output.

```
export class Result {
  fileName: string;
  topLevel: TopLevelPropertiesResult;
  methods: MethodsResult;
  bindings: BindingsResult;
  ...
}
```

4.3 Interaction Diagram Generation

The generation of the interaction diagram graph from the result class from 4.2.5 is done in the *Transformer* class.

The resolution of methods is abstracted in the *MethodResolver* class. It produces a *ResolvedMethodDefintition* for each called method in bindings and the initial method. In order to prevent duplicate resolution of methods and wasting of resources a *MethodCache* is introduced. The *Transformer* does not use the *MethodResolver* directly, but instead accesses it via the *MethodCache*. The cache includes directly called (bound to) and indirectly called (calls of methods), for each of which vertices will have to be created. Each *ResolvedMethodDefintition* has the following data type:

```
export enum GeneralisedArgument {
  METHOD = "method",OTHER = "other",
}
export type ResolvedArgument =
  | Property | GeneralisedArgument.METHOD | GeneralisedArgument.OTHER;
export interface ResolvedMethodDefintition {
  id: Identifiers;
  args: ReadonlyArray<ResolvedArgument>;
  reads: ReadonlyArray<Property>;
  writes: ReadonlyArray<Property>;
  calls: ReadonlyArray<CalledMethod>;
}
export interface CalledMethod {
  id: Identifiers;
  args: ReadonlyArray<ResolvedArgument>;
}
```

As the underlying structure for the graph graphlib is used and wrapped in an own class - *ExtendedGraph*. It creates vertices on a 'create if not exist' basis by first looking up to see if the vertex exists in the graph, and if it does, does not add it again. Presence of edges is not checked, if an edge is added again, the previous one is simply overwritten. There can only be one edge per direction between two nodes, since no multigraph is used. Nodes and Edges in the graph have the following structure, as specified in 3.2:

```
export enum EdgeType {
  SIMPLE = "simple", EVENT = "event", CALLS = "calls",
}
export interface Edge {
  source: Node;
  sink: Node;
  label: EdgeType;
```

```
}
```

```
export enum NodeType {
  TAG = "tag",  DATA = "data", METHOD = "method", INIT = "init",
}

export type Node = TagNode | DataNode | MethodNode | InitNode;
interface BaseNode {
  id: string;
  name: string;
}
export interface TagNode extends BaseNode {
  loc: Location;
  discriminator: NodeType.TAG;
}
export interface MethodNode extends BaseNode {
  discriminator: NodeType.METHOD;
}
export interface InitNode extends BaseNode {
  discriminator: NodeType.INIT;
}
export interface DataNode extends BaseNode {
  parent?: string;
  type: IdentifierType;
  discriminator: NodeType.DATA;
}
```

The algorithm for generating the interaction diagrams graph as described in 3.2.4 is implemented in the *Transformer* class.

First the init method is resolved by querying the *MethodCache* and afterwards each of the bindings. The order is not important. If the bindings are properties, the vertices for them are created directly. If they are methods or computed properties, they are resolved by the *MethodCache* and the correct edges based on the binding type created. The above is done in the *addInit()* and *addBindings()* methods.

At this stage no vertices are created for the reads, writes, and calls properties of the resolved methods. This happens in the *addIndirectlyCalledMethods()* method, after all bindings and the init method have been resolved by taking all methods stored in the *MethodCache* and creating the appropriate vertices.

Lastly in the *addEdgesForLists* method additional edges will be added for the *all* vertices of properties of elements inside lists.

4.4 Scenario Generation

The scenario generation is implemented in the *scenario.ts* file and currently ouputs the obtained scenarios, tags, nodes react to and the gherkin scenarios templates to the console. The scenarios are generated exactly as described in 3.3

4.5 Usage

The application can be started using

```
npm run generate -- [file to parse] [output graph path] [depth]
```

where file to parse is the path to the *.vue* file to be parsed and output graph path is the path to which the output interaction diagram graph will be written (JSON object). Depth is the factor k in 3.3. A web server can be startet in order to view the results in a browser at *localhost:8000* via

```
npm run serve
```

In order to create a snapshot for each file in *resources/test-files*.

```
npm run create-results
```

Can be used. This snapshot includes the output interaction diagram graph as *data.json*, the generated scenarios as a *scenario.txt*, *index.html*, which can be used to display the interaction diagram and a *.vue* file for which all of the above was generated.

All snapshots can be viewed using

```
npm run results
```

and navigating to *localhost:8001*

Chapter 5

Testing and Evaluation

1. zhang code, generated 2. example with lists 3. complex list(add, sub) with focus on updates

Chapter 6

Conclusion

- lists recursive - use directly the ast of eslint for true solution
- betterr output not just console

Appendix A

Appendix

TODO code attached yada yada

List of Figures

2.1 MVVM design pattern overview, adapted from [Bri17, p. 7] 10

2.2 ESLint Architecture taken from [21c] 12

3.1 AST for top-level example 17

3.2 AST for example 18

3.3 AST for example 19

3.4 Example Graph obtained for the identifier *this.problems[0].a* 20

3.5 Example Graph for the following object accesses - *this.problem.a*, *this.problem.b*, *this.problem.author.name* 21

3.6 Generic List representation 22

3.7 Concrete example of a list representation 22

3.8 Example Graph including list elements with properties 24

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`.

[21e] *Lodash*. [Online; accessed 9. Feb. 2021]. Feb. 2021. URL: `https://lodash.com`.

[dag21] dagrejs. *graphlib*. [Online; accessed 9. Feb. 2021]. Feb. 2021. URL: `https://github.com/dagrejs/graphlib`.

[21f] *Babel - The compiler for next generation JavaScript*. [Online; accessed 9. Feb. 2021]. Feb. 2021. URL: `https://babeljs.io`.

[Jac21] Magnus Jacobsson. *d3-graphviz*. [Online; accessed 9. Feb. 2021]. Feb. 2021. URL: `https://github.com/magjac/d3-graphviz`.

[Che21] Tianxiang Chen. *light-server*. [Online; accessed 9. Feb. 2021]. Feb. 2021. URL: `https://github.com/txchen/light-server`.