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)
Department 4

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 11, 2021

Abstract

This thesis aims to further explore and expand on the concept of Interaction Diagrams for scenario testing of single-page applications (SPAs) introduced by [ZZ19] for applications written using the Vue.js framework. Interaction Diagrams model the reading, writing and invocation relationships between HTML tags as a directed graph in order to visualize the possible work flows of the application. An approach to automatically generate this diagrams from static Vue.js source code is presented including support for simple lists and complex objects. It is also shown how the diagrams can be used in order to generate scenario templates in the Gherkin domain-specific language (DSL), which can be used in behavior-driven Development (BDD).

Contents

1 Introduction **5**

2 Fundamentals **6**

2.1 Related Work 6

2.2 Scenario Testing of AngularJS-based Single Page Web Applications 6

2.2.1 Abstract Syntax 7

2.2.2 Interaction Diagrams 7

2.2.3 Testing and Interactions 8

2.2.4 Coverage Criteria 8

2.3 Scenario Testing 9

2.4 Behavior-Driven Development 9

2.4.1 Gherkin Language 10

2.5 Model-View-ViewModel 10

2.5.1 View 10

2.5.2 View-Model 10

2.5.3 Model 10

2.6 Vue.js 11

2.6.1 Components 11

2.6.2 Data Binding 11

2.7 ESLint 12

2.7.1 Rules 12

2.7.2 Selectors 12

2.7.3 AST Explorer 13

2.7.4 ESTree AST 13

2.7.5 ESLint Parser Vue AST 13

3 Concept **14**

3.1 Parsing Vue.js 14

3.1.1 Assumptions 14

3.1.2 Limitations 14

3.1.3 AST 15

3.2 Interaction Diagram Generation . . . 18
3.2.1 Variable Identifiers . . . 18
3.2.2 Object representation . . . 19
3.2.3 List representation . . . 20
3.2.4 Method representation . . . 21
3.3 Scenario Generation . . . 23

4 Implementation **25**
4.1 Project Structure and Overview . . . 25
4.1.1 Project structure . . . 25
4.2 Parsing Vue.js . . . 26
4.2.1 Common Data Types . . . 26
4.2.2 Top Level Properties . . . 27
4.2.3 Bindings . . . 27
4.2.4 Method Definitions . . . 29
4.2.5 Output . . . 30
4.3 Interaction Diagram Generation . . . 30
4.4 Scenario Generation . . . 32
4.5 Display of Interaction Diagrams . . . 32
4.6 Usage . . . 32

5 Testing and Evaluation **34**
5.1 Math for Kids - Simple Properties . . . 34
5.1.1 Scenarios . . . 36
5.2 Math for Kids Extended - Lists and Computed Properties . . . 37
5.3 Menu with daily meal - Lists and Objects . . . 39

6 Conclusion **42**

A Appendix **47**
A.1 Submitted Source Code . . . 47
A.2 ESLint . . . 47
A.2.1 Notable AST Nodes . . . 47
A.3 Source Code . . . 49
A.3.1 Math for Kids . . . 49
A.3.2 Math for Kids Extended . . . 51
A.3.3 Menu with daily meal . . . 55
A.4 Generated Gherkin Scenarios . . . 57
A.4.1 Math for Kids Extended . . . 57

Chapter 1

Introduction

Vue.js [Vue21a] is a popular progressive frontend framework for building user interfaces and single-page applications. Automatic test generation is a fascinating problem in Computer Science, especially for dynamically typed languages such as JavaScript. It presents a series of benefits, predominantly conserving of development time and allowing for more focus on the development of the main application. Interaction Diagrams can also aid the understand of SPAs, since in SPAs there is no clear navigation, rather contents are updated dynamically based on user input.

The focus of this thesis lies on the implementation of an algorithm to generate Interaction Diagrams, as described by Zhang and Zhao [ZZ19]. The approach by Zhang and Zhao [ZZ19] is also extended to also be able to model lists and complex objects. Further it is shown how the generated diagrams can be used in order to generate scenario templates in the Gherkin DSL, which can be used in BDD.

The thesis is structured as follows:

First fundamental technologies will be explained in Chapter 2. Then in Chapter 3 the concept of the algorithms will be described, followed by Chapter 4, which will delve deeper into some implementation details. Finally the application will be evaluated in Chapter 5, followed by the conclusion 6.

Chapter 2

Fundamentals

2.1 Related Work

Andreasen et al. [And+17] survey automated test generation in section 8 of their work[And+17, pp. 23–25]. They survey a total of 14 papers in this category and separate them into one of three approaches - explorations of the event space (concerning the order of execution of event handlers), explorations of the value space (choice of values for inputs), and generation of assertions [And+17]. In the event space category their findings include approaches based on random executing of event handlers, heuristic methods, based on observations of read and written values in order to exclude handlers with no interaction, and a tool for measuring the performance impact of handlers. Most of their findings in the value category focus on concolic execution [GKS05], a technique for generating concrete inputs (test cases) in order to maximize test coverage. Almost all of their finding in the assertion generation category are based on Crawljax [cra21], an event-driven open source web crawler for dynamic Ajax-based Web applications, which is also capable of exploring JavaScript-based Document Object Model (DOM) state changes. A table with a more detailed overview can be found in [And+17, p. 24].

Model checking of Web applications - a method for checking whether a finite-state model of a system conforms to a specification, is explored in [ZZ19] and [Gao+19]

2.2 Scenario Testing of AngularJS-based Single Page Web Applications

Zhang and Zhao [ZZ19] present a method with the goal of achieving better understanding of AngularJS-based SPAs and also devise 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 an 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 is 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).

Nodes

The set of nodes N is defined as the union of N_H (HTML tag nodes), $N_{\$scope}$ (databindings and events) and N_{js} (JavaScript functions).

Formally $N_H = \{n_h | (h, v) \in D\}$, $N_{\$scope} = \{n_v | (h, v) \in D\} \cup \{n_e | (h, e) \in E\}$ $N_{js} = \{n_v | v \in W\} \cup \{n_f | f \in F\}$ The initialization node, n_{init}, is distinguished by an incoming arrow without a starting vertex.

Edges

The set of edges E is a combination of seven sets:

- data bindings are defined as $e_d = (target(d), source(d))$, $E_{data} = \{e_d | d \in D\}$. Additionally for two-way bindings: if $d \in D'$ also create $e'_t = (source(t), target(t))$ and $E'_{data} = \{e_d | d \in D'\}$
- Events are defined as (E_{event}) $(h, f) \in E$
- Variables methods write to - E_W, (f, v) where $f \in F \cup F(\$scope), v \in W(f)$
- Variables methods read from - E_R (v, f) where $f \in F \cup F(\$scope), v \in R(f)$

- Methods, invoked by other methods - E_{Inv} (f, v) where $f \in F \cup F(\$scope), v \in I(f)$
- default values of widget set by the *init* function E_{init} default values of widgets for each $h \in T$ where $\nexists v|(h, v) \in D$ create an edge (h, v)

Zhang and Zhao [ZZ19, p. 9] provide a more detailed explanation of the above, including examples.

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 Interaction Diagrams as described in 2.2.2, it is possible to derive which widgets get updated by a user input action or setup 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 sense, 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 interactions.

A sequence of user interactions, including the initial function is referred 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 reacted to - $l(A) = l(a_n)$.

The set of scenarios is generated by starting with the initial scenario, containing only the initial function $S_0 = \{(init)\}$ and prolonged 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 - 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 its 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 succeeded 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) and exposing of 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].

The main goal of BDD is to specify a system in terms of its functionality (i.e. its 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, specifying the prerequisite/context, event and outcomes of a scenario, respectively. Typically scenarios are written by the *The Three Amigos* - the product owner, tester and developer [21a].

2.4.1 Gherkin Language

Gherkin [21b] is a DSL for definition of test scenarios using the BDD verbs, described in 2.4. It is used in the Cucumber testing framework, which provides tools to generate tests templates in various programming languages.

2.5 Model-View-ViewModel

Model-View-ViewModel (MVVM) is a design pattern, which helps in creating a clear separation between business 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.

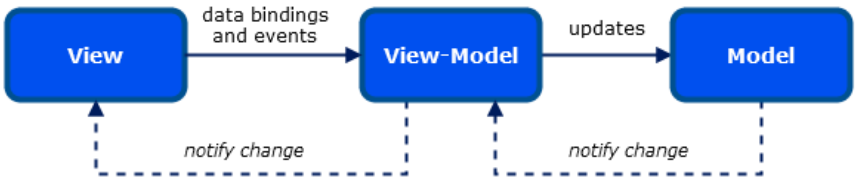


Figure 2.1: MVVM design pattern overview, adapted from [Bri17, p. 7]

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 of its logic.

2.6 Vue.js

Vue.js [Vue21a] is a progressive front end framework for building user interfaces and single-page applications based on the MVVM design pattern described in 2.5 [Mac18] [21c].

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 [21c].

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>`. The `template` is a HTML-based template, which can be parsed by specification compliant browsers and HTML parsers. It can contain other components or HTML elements and is equivalent to the *view* in MVVM.

The `script` section of a Vue.js component includes the view-model of the component. Each component has a special JSON object `data`, equivalent to the MVVM model. The `script` part of a computed includes CSS-like styles.

2.6.2 Data Binding

Vue.js has built in support for data binding using special directives.

One-way Binding

One-way binding is a one directional binding from source - a property of the model or other components to target - a property of a component or HTML tag. It can be achieved via the `v−bind` or *moustache* syntax (double curly brackets). One-way bindings can also contain complex expressions.

Two-way Binding

Two-way binding is similar to one way-binding, but the value, which is bound to goes both ways and is synchronization. A typical example is a binding from an input field to a property of the model, e.g `email`. It can be achieved via the `v−model` binding directive.

Event binding

Event bindings are bindings of methods of the view model to events of components or HTML tags, for example the click of a button. It is done via the `v−on` or `@` directive. Event bindings can also contain anonymous methods.

Computed Properties

Computed properties are special properties, which can be used in data binding and are a composition of other properties.

V-for Binding

A `v-for` directive is a special type of binding used to render lists, which semantically is used similarly to a `foreach` loop.

2.7 ESLint

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

2.7.1 Rules

At the core of ESLint are rules, which are extensible pieces of code, bundled as plugins, that can be used to verify various aspects of code. An example would be a rule, which checks for matching closing parenthesis. 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 whether 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 is the implementation of the rule and 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 AST.

2.7.2 Selectors

ESLint provides a very powerful matching mechanism for specifying what nodes to visit called selectors [21e]. It is inspired by estools [est21].

Some notable selectors include:

- AST Nodes `MemberExpression` - matches any node of type `MemberExpression`
- descendant `MemberExpression Identifier` - matches any node of type `Identifier` who has a descendant of type `MemberExpression` in the tree
- child `MemberExpression > Identifier` - matches any node of type `Identifier` who has a direct parent (is a child of) of type `MemberExpression`

- :not `:not(MemberExpression)` - negation of a selector. In this case, matches any node, that is not a `MemberExpression` node.
- :matches - `:matches(MemberExpression, Identifier)` matches any of the selectors. In this case, either a `MemberExpression` or `Identifier` .
- attributes `Property[name=a]` . - Matches any AST Node, whose specified attribute has the specified value.

2.7.3 AST Explorer

An incredibly useful tool when working with ASTs is AST Explorer, developed by Kling [Kli21]. It enables the exploration of syntax trees generated by various parsers and also includes the vue-eslint-parser [Vue21b].

2.7.4 ESTree AST

By default ESLint uses the Espree [ESL21] parser to parse JavaScript source code into an AST defined in the ESTree specification [EST21]. When Parsing `.vue` files ESLint uses this parser for the code inside the `<script>` tag.

2.7.5 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 AST specification, 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 based on an AST, which covers every possible syntax, such as [Vue21c]. 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)
- computed properties
- complex objects
- non-nested lists
- methods, including the resolution of arguments, they have been called with (excluding 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: accessedVariable*;
writes: accessedVariable*;
calls: calledMethod*;

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

bindings: binding*;
binding: tag bindingSource+;
bindingSource: (accessedVariable | calledMethod) (EVENT_BINDING | ONE_WAY_BINDING) |
    accessedVariable 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 model. For example `problem:{a:0, b:0}` will be represented as follows:

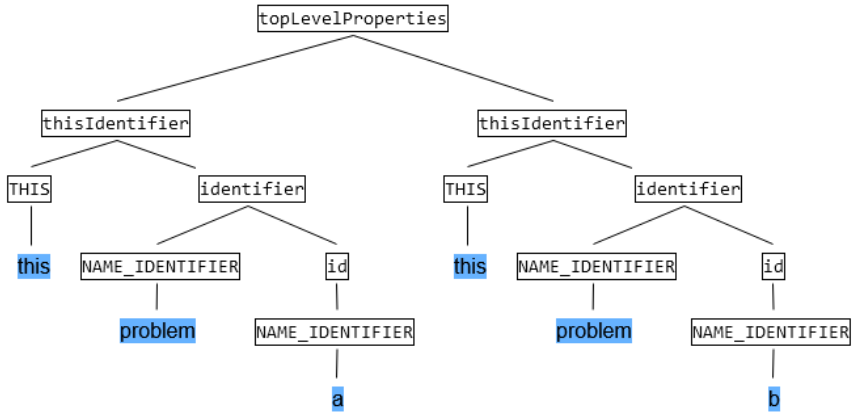


Figure 3.1: AST for top-level properties example

The bindings of the Vue.js component (`bindings`) can be obtained from the `template`. Each binding consists of an HTML tag, a list of binding sources for that tag - pairs of variable or method calls 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 && func()"/>`). 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 (`calledMethod`) include the arguments 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 arguments - 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 AST:

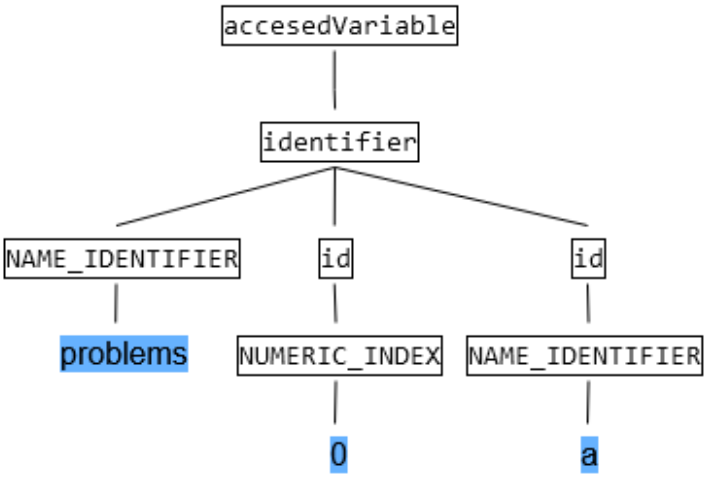


Figure 3.2: AST for a list access example

Another special case are `v–for` statements. They 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:

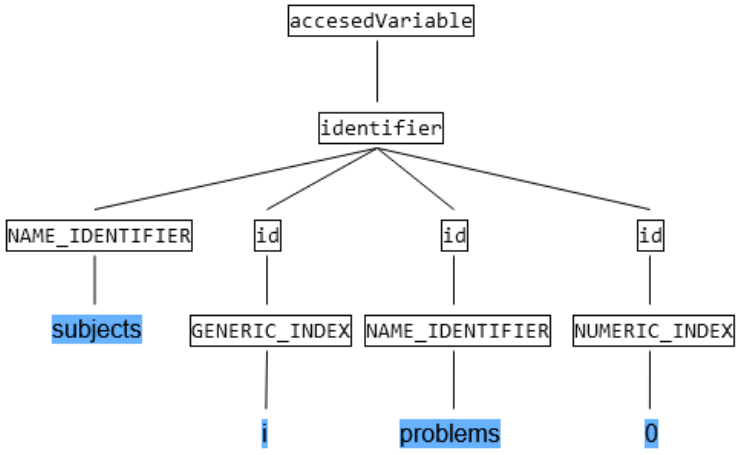


Figure 3.3: AST for a `v–for` substitution example

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` - variables it writes to
- `calls` - method calls, including arguments, same as for bindings

Computed properties (`computedProperties`) are similar to `methodDefinitions` with the exception that they do not have arguments. Albeit bad practice, 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 3.1.3 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, compound graph will be leveraged and later on converted to a directed graph. 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 required in order to add additional edges for lists.

Vertices in this graph will 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 vertices. 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'.

3.2.1 Variable Identifiers

Variable identifiers are represented by `identifier` and `thisIdentifier` in the AST 3.1.3. For the `THOS` and for each `id` node in the `identifier` or `thisIdentifier` a vertex is created in order. Those vertices are connected using unidirectional 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 written to from a method (`write`), nodes of type `GENERIC_INDEX` are omitted. The reason behind this will be explained in 3.2.3;

Each vertex has a GUID equal to the value of its terminal symbols - `NUMERIC_INDEX`, `GENERIC_INDEX` or `NAME_IDENTIFIER`, concatenated with the value of the previous vertex. The `label` of those vertices are equal to the terminal symbol in case of `NAME_IDENTIFIER`. In case of `GENERIC_INDEX` and `NUMERIC_INDEX`, they are 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.

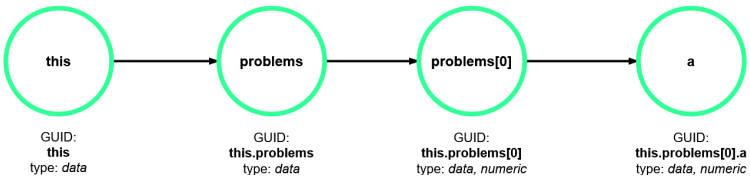


Figure 3.4: Example Graph obtained for the identifier `this.problems[0].a`

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`.

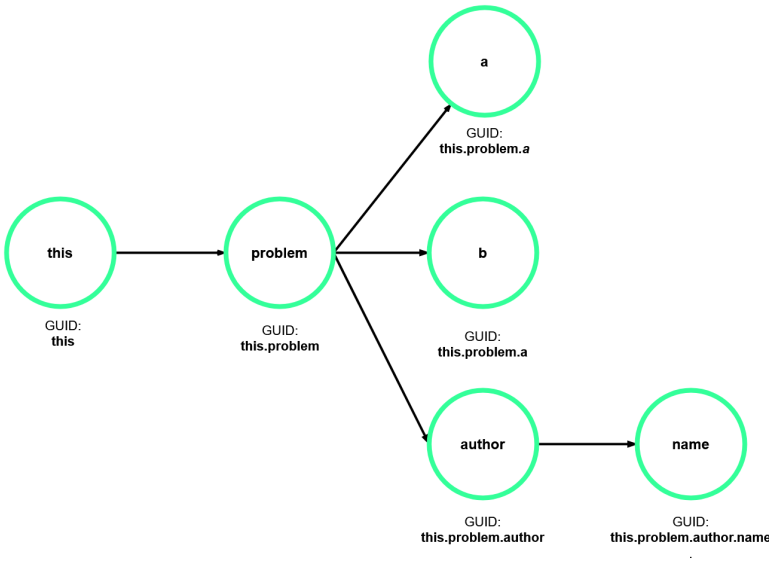


Figure 3.5: Example Graph for the following object accesses - `this.problem.a`, `this.problem.b`, `this.problem.author.name`

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 change, it would only result in a cascading change in `name`.

3.2.3 List representation

Fig. 3.8 shows the abstract idea behind list representations.Concrete elements, which are accessed, are denoted 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, ... 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, ... P_n, however in an update to P_any. 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 and an example can be seen later in section 3.8.

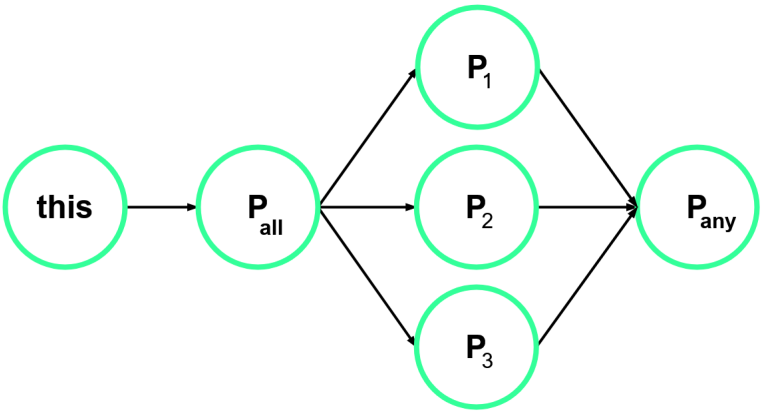


Figure 3.6: Abstract list representation

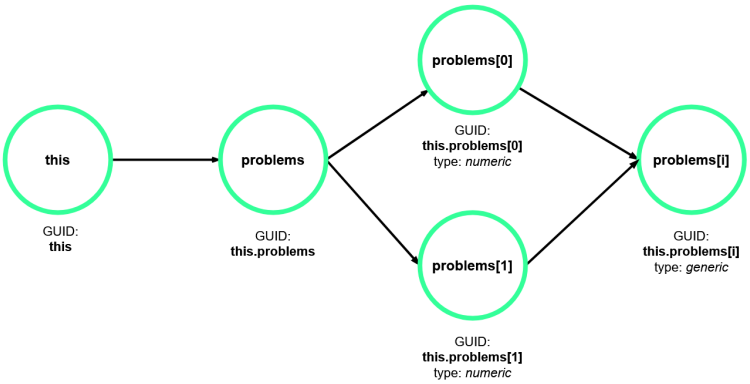


Figure 3.7: Concrete example of a list representation

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 successful 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 concatenated 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` referencing 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 substitution 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.

Generating Interactions

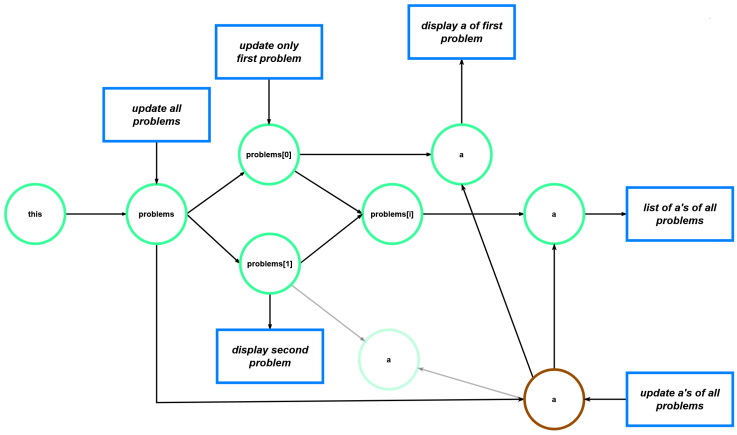


Figure 3.8: Example Interaction Diagram graph including list elements with properties. Circled nodes represent data node and rectangular ones HTML tags and methods.

Interaction diagrams can be generated from the simplified Vue.js AST in the following way: For each `binding` in `bindings` and for each `tag`, `bindingSource` in `binding` create a vertex for `tag`, with a GUID `tagId`, 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 is as top level property, by doing a lookup on `topLevelProperties`, treat it as a property and create vertices for it as described 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 a `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, 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, an edge with the binding vertex as source and the tag vertex as sink.
- C) If the binding type is two-way, 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.

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 additionally other properties (properties on list elements, with $type(v) \neq$ 'generic', $type(v) \neq$ 'numeric'). For each of those parent vertices p 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.

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 denote 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 referred 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 additionally $n \in l(A)$ 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 template:

$$
\begin{aligned}
Scenario &: n_0, n_1 \dots n_k \\
Given &: n_0, n_1 \dots n_{k-1} \\
When &: n_k \\
Then &: l(n_k)
\end{aligned}
$$

Chapter 4

Implementation

4.1 Project Structure and Overview

The application is written in TypeScript(excluding the ESLint visitors, which have to be in JavaScript) using NPM as a package manager and Node.js as a runtime environment. Notable dependencies include TypeScript[21f], for stricter syntax and types, lodash[21g] for enrichening of collections, graphlib[dag21] for the interaction diagrams graph, babel[21h] as a transcompiler, ESLint[21d], Estree[EST21] and eslint-plugin-vue[Vue21b] for parsing Vue.js code, d3-graphviz[Jac21] in combination 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.

The application source code is linted using ESLint, unit tests are written in Jest (total of 109 tests cases) and also includes simplistic regression tests. Git is used for version control and also a CI/CD pipelines is setup, which on every commit executes the tests and regression tests and also ensures there are no type errors, as reported by TypeScript, and no linting errors, as reported by ESLint. A separate minimal Vue.js project, in which the test Vue.js SPA applications can be rendered and easily viewed in a browser, was implemented as a separate repository and can be found here [Kar21].

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, There is also a `common` package, which is shared among all. Each will be described in more detail in the following sections. Each package includes a models directory, which includes 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 at different timestamps (with the latest being the current) and `resources` hosts various additional resource files and outputs.

```
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 simplified Vue.js AST described in 3.1.3 the capabilities of ESLint [21d] 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 definitions, computed properties and the created method.

4.2.1 Common Data Types

Below are the common data types shared among all visitors. Discriminators are used to be able to differentiate 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.

```
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;
}
```

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` A.2.1. It can be selected via the following selector:

```
"ExportDefaultDeclaration > ObjectExpression >
Property[key.name = data] ReturnStatement > ObjectExpression"
```

In natural language the selector reads: "select `ObjectExpression` nodes, which have a direct parent `ReturnStatement` node, 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 A.2.1 of the `ObjectExpression` the name of the key (identifier) is stored. If the property is an object (value of `ObjectExpression`) 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` A.2.1, `Identifier` A.2.1, `MemberExpression` A.2.1 and `CallExpression` A.2.1.

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` A.2.1 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` A.2.1. 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` A.2.1. There does not seem be an easy way to select

all properties read from. Therefore all accessed 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

Combining 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 `ResolvedMethodDefinition` 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 `ResolvedMethodDefinition` has the following data type:

```
export enum GeneralisedArgument {
  METHOD = "method",OTHER = "other",
}
export type ResolvedArgument =
  | Property | GeneralisedArgument.METHOD | GeneralisedArgument.OTHER;
export interface ResolvedMethodDefinition {
  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 outputs the obtained scenarios, tags, nodes react to and the Gherkin scenario templates to the console. The scenarios are generated exactly as described in 3.3.

4.5 Display of Interaction Diagrams

In order to display the interaction diagrams it is displayed on a single HTML page, utilizing the d3-graphviz [Jac21] library.

Event edges are displayed as thick bold lines, calls edges are displayed as solid lines and all other as dashed lines.

HTML tag nodes s are separated from all other nodes in a subgraph in order to be able to more easily identify them. In addition to their name, they also include the line(s) on which they are in the source code as a subline. The initial function has a grey background so it can be identified right away. A button to download the displayed diagram as a `.png` image is also provided in the top left.

4.6 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 started 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
```

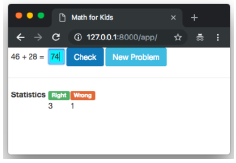
and pointing the browser to `localhost:8001`.

Chapter 5

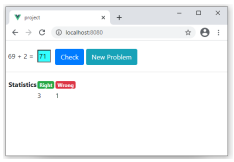
Testing and Evaluation

5.1 Math for Kids - Simple Properties

Zhang and Zhao [ZZ19] use a SPA titled "Math for kids", which generates simple math addition problems and keeps track of a statistic, which includes the number of correctly and incorrectly answered problems as an example. The same application has been implemented and extended in Vue.js and will be used to showcase the capabilities of the interaction diagram and scenario generation application and compare the resulting diagrams to the ones in [ZZ19].



(a) Math for Kids in AngularJS by Zhang and Zhao [ZZ19]



(b) Own implementation of Math for Kids in Vue.js

The source code can be found in `resources/test-files/test.vue` and also in A.3.1.

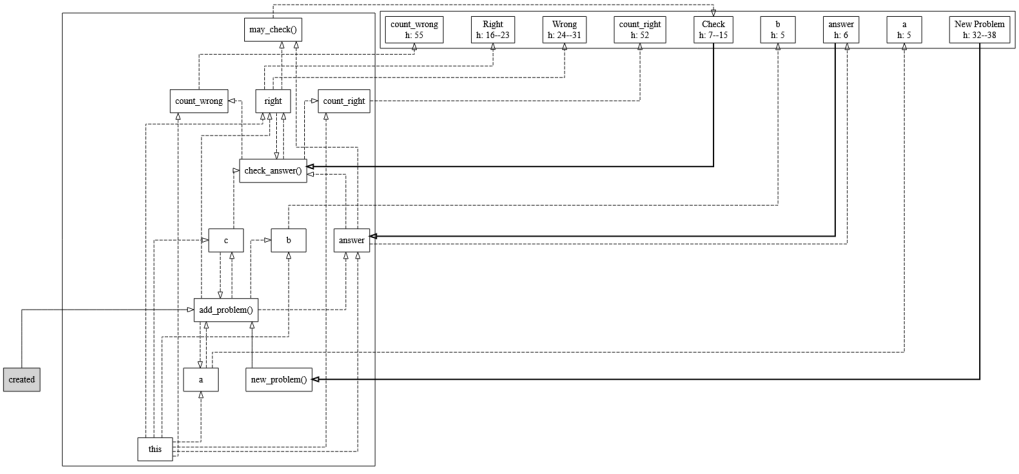


Figure 5.2: Math for Kids in Vue.js generated interaction diagram

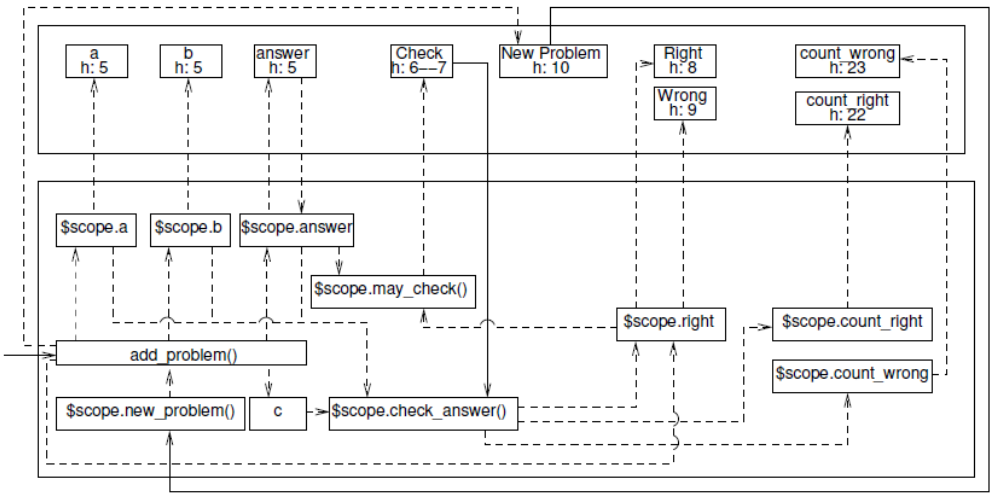


Figure 5.3: Math for Kids in AngularJS interaction diagram by Zhang and Zhao [ZZ19]

When comparing the generated diagram 5.2 to the original 5.3 by Zhang and Zhao [ZZ19] there are some differences.

Due to different modelling there is a `this` vertex and connections from it to top level variables, which does not exist in the diagram by Zhang and Zhao [ZZ19]. In the generated diagram there is an edge of type 'event' between the `answer` tag and `answer` property. This is due to the difference in two-way bindings representation.

There are two differences between the `add_problem` node in the generated diagram and the one by [ZZ19]. In Zhang and Zhao [ZZ19] there is an edge from the `add_problem` method to the `New Problem` tag, which is missing in the own generated version. This seems like a mistake in [ZZ19], since this edge should not exist.

The other difference is that in [ZZ19] there is no edges from `add_problem` to `right` (write relation), however there should be, since inside `add_problem`, `this.right = undefined`.

```
l(created) -> a, b, answer, Check, Right, Wrong
l(answer) -> answer, Check
l(Check) -> Right, Wrong, Check, count_right, count_wrong
l(New Problem) -> a, b, answer, Check, Right, Wrong
```

The generated interactions differ for $l(created)$, all other sets are exactly the same as the ones by Zhang and Zhao [ZZ19] (albeit in different order). This may stem from the fact that the edges of `add_problem` to `$scope.right` is missing in [ZZ19], however it is later correctly factored in l(New Problem).

Zhang and Zhao [ZZ19] define the initial interaction as `l(add_problem()= {a, b}` whereas it should be `l(add_problem()= {Check, Right, Wrong, a, b, answer }` since the answer property is updated based on the diagram.

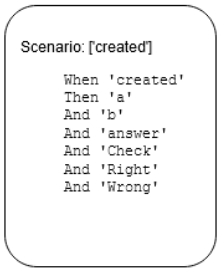
One could argue, that the version in [ZZ19] is correct, since *answer* is set to *undefined*, which is the same as the initial value of the variable, so it would not trigger an update as part of the `init` method. The interaction diagram generator does not perform this check. If `add_problem()` were

to be called later in the application (when the `New Problem` button is clicked) it would indeed set a new value to `answer`, which is correctly reflected by Zhang and Zhao [ZZ19].

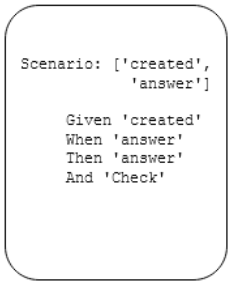
5.1.1 Scenarios

The generated Gherkin scenario templates of up to 4 actions can be seen in 5.4. The program outputs the scenarios as plain text to the console, but here they are displayed in a nicer way. The caption of each figure is an example of how a text could be generated based on the template scenario output by the application. Some scenarios seem a bit repetitive, but it is up to *The Three Amigos* 2.4 to decide which templates to discard, since interaction diagrams model what *might* be updated. The fact that everything which might get updated is displayed can also be leveraged in another way - negative criteria can be defined (verify component or tag X did not change).

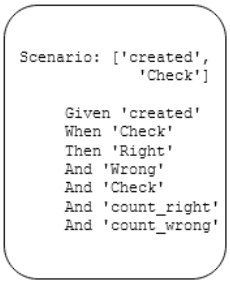
Figure 5.4: Gherkin scenario templates and sample human written scenarios based on the templates



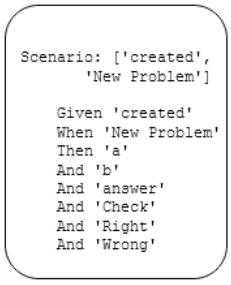
(a) Scenario: initialization. When the application is created, then 'a' and 'b' should show random numbers and 'Check' should be disabled and 'Right' and Wrong should be invisible.



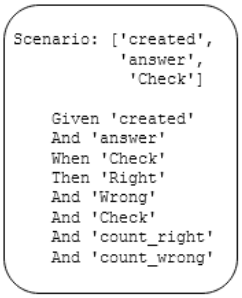
(b) Scenario: typing an answer. Given the application has been created, when I type an answer then 'answer' should display it and 'Check' should be enabled.



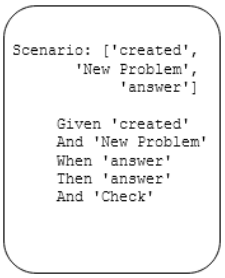
(c) Scenario: clicking on check without typing an answer. Given the application has been created, when I click 'Check' then 'Right' and 'Wrong' should be invisible and 'count_right' and 'count_wrong' should have the same values and 'Check' should be disabled.



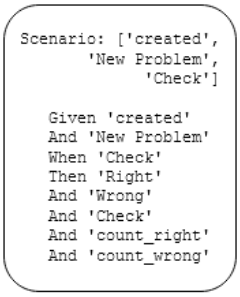
(d) Scenario: obtaining a new problem at application start. Given the application has been created, when I click 'New Problem' then 'a' and 'b' should have random values and 'answer' should be empty and 'Check' should be disabled and 'Right' and 'Wrong' should be invisible.



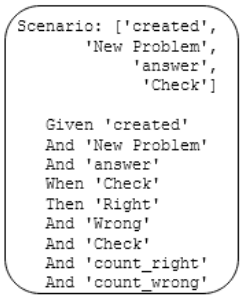
(e) Scenario: checking my answer. Given the application has been created and 'answer' contains my answer, when I click 'Check' then 'Right' should be visible if the answer was right 'Wrong' should be visible if the answer was wrong and 'Check' should be disabled and 'count_right' should be incremented by one if the answer was right and 'count_wrong' should be incremented by one if the answer did not change.



(f) Scenario: typing an answer to a new problem. Given the application has been created and a new problem was obtained, when I type an answer then 'answer' should display it and 'Check' should be enabled.



(g) Scenario: requesting a new problem and clicking on check without answering it. Given the application has been created and I requested a new problem, when I click 'Check' then 'Right' and 'Wrong' should be invisible and 'count_right' and 'count_wrong' should have the same values and 'Check' should be disabled.



(h) Scenario: requesting a new problem, answering it and checking my answer. Given the application has been created and I requested a new problem and answered it, when I click 'Check' then 'Right' should be visible if the answer was right 'Wrong' should be visible if the answer was wrong and 'Check' should be disabled and 'count_right' should be incremented by one if the answer was right and 'count_wrong' should be incremented by one if the answer did not change.

5.2 Math for Kids Extended - Lists and Computed Properties

The Math for Kids application has been extended to display a list of past problems and randomly generates either a subtraction or addition problem. It also keeps tracks of statistics separated by problem type and includes and additional accuracy statistic, which is implemented as a computed property. The application can be seen in 5.5 and the full source code can be found it A.3.2

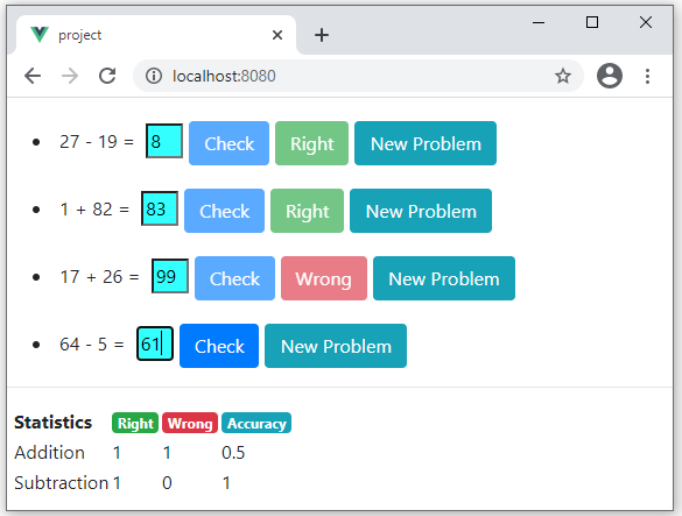


Figure 5.5: Math for Kids in Vue.js including subtraction problems, more precise statistics and display of past problems

The resulting interaction diagram is shown in 5.6. It looks more complex, but it is still possible to focus on specific components and see how they interact with others.

The two vertices for the computed properties *accuracy_sub* and *accuracy_add* correctly directly depend on *count_right_sub*, *count_wrong_sub* and *count_right_add*, *count_wrong_add* respectively. By looking at the interactions, one can observe that based on which of the mutually exclusive Check buttons is clicked on, either the substituting or addition related properties are updated. This differentiation is also evident in the generated Gherkin scenario templates. The full list of generated scenario templates of up to 4 actions can be seen in A.4.1

```
l(problems[i].answer) -> problems[i].answer, Check, Check
l(Check) -> Right, Wrong, Check, Check, count_right_add, accuracy_add,
    count_wrong_add
l(Check) -> Right, Wrong, Check, Check, count_right_sub, accuracy_sub,
    count_wrong_sub
l(New Problem) -> problems[i].a, +, -, Check, Check, problems[i].b,
    problems[i].answer, Right, Wrong
l(created) -> problems[i].a, +, -, Check, Check, problems[i].b,
    problems[i].answer, Right, Wrong
```

This example includes a bit of engineering - more realistically there would a single button, which checks inside the bound method which should be updated, however the application would not be able to be differentiate them in this case, since if statements inside methods are not supported.

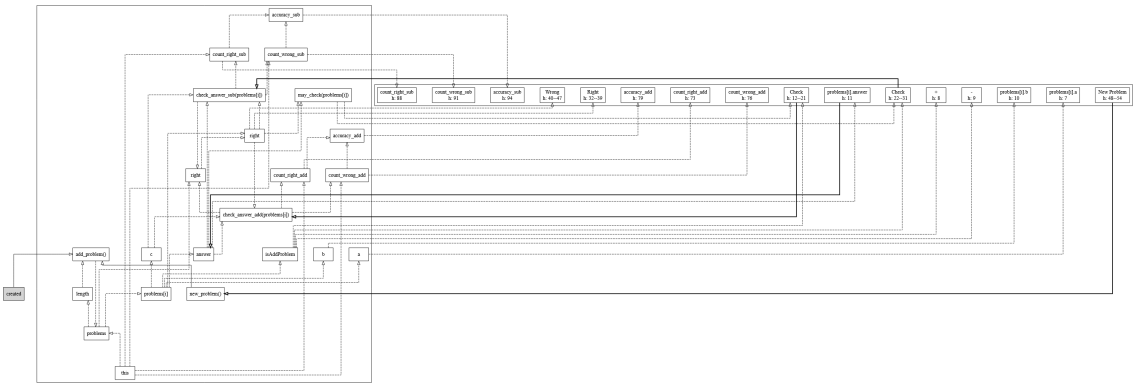
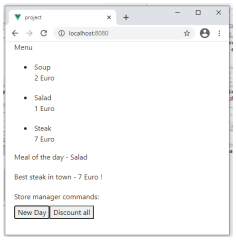


Figure 5.6: Generated interaction diagram for Math for Kids including subtraction in Vue.js

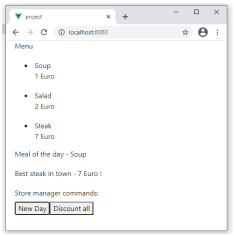
5.3 Menu with daily meal - Lists and Objects

Another simple example SPA with the aim of evaluating the capabilities of the application when dealing with lists and complex objects was devised and can be seen in 5.7. It shows a which displays a minimalistic menu for a restaurant with a meal of the day. The full source code for it can be found in A.3.3.

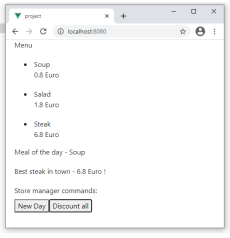
The daily meal and its price is changed every day via a button click, and alternates between the first two items. The third item, steak, is never the daily meal. There is also an option to discount the price of all items. With stakes being the restaurant's special, there is an additional advertisement label them.



(a) The current meal of the day is salad.



(b) The 'new day' button has been clicked and the daily meal now changed to soup.



(c) Discount was clicked and the price of all items got updated.

Figure 5.7: The menu with daily meal application

Fig. 5.8 shows the interaction diagram for the application. The resulting interactions can be

seen in 5.3 and scenarios of up to 4 steps are shown in 5.3.

```
l(New Day) -> Price, Meal of the day -
l(Discount all) -> Best steak in town -, Price
l(created) -> meals[i].name, Price, Best steak in town -, Meal of the day -
```

```
Scenario: ['created']
        When 'created'
        Then 'meals[i].name'
        And 'Price'
        And 'Best steak in town -'
        And 'Meal of the day -'

Scenario: ['created', 'New Day']
        Given 'created'
        When 'New Day'
        Then 'Price'
        And 'Meal of the day -'

Scenario: ['created', 'Discount all']
        Given 'created'
        When 'Discount all'
        Then 'Best steak in town -'
        And 'Price'
```

The `New Day` button changes the `Meal of the day` and some/all of the `Price` list HTML tags. Note that only the price of each meal is changed, the `meals[i].name` HTML tags remain the same. `Best steak in town` is also not updated, since steak, the last element, is the daily meal.

Clicking on `Discount all` does indeed change both `Best steak in town` and some/all `Price` HTML tags, since everything is discounted.

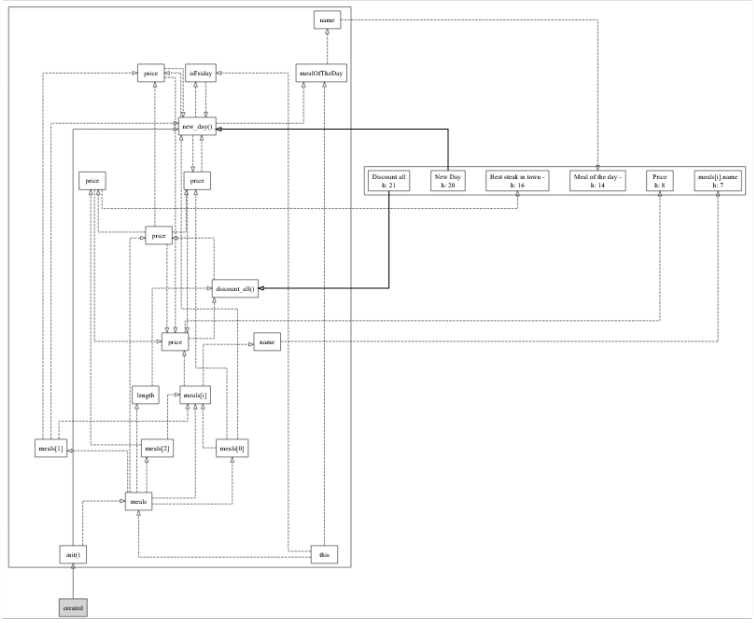


Figure 5.8: Generated interaction diagram for Menu with daily meal

Chapter 6

Conclusion

In this thesis an algorithm to generate interaction diagrams as described by Zhang and Zhao [ZZ19] for Vue.js was implemented and extended with support for lists objects and objects. Further it was also shown how the interaction diagrams can be used to automatically generate Gherkin test scenario templates

The application is able to correctly generate interaction diagrams and scenario templates, which can be used to visualize the work flow of a SPA. The scenario templates can successfully be used as an aid to generate very extensive Gherkin test scenarios.

The application can be extended by more language constructs and features such as if-else statements, nested lists, arrow functions. In order to fully capture the complexity of JavaScript it must be extended to operate directly on a complete AST, such as the one generated by the ESLint parser. A proper user interface for the Gherkin templates could also be implemented or it could be integrated as a plugin into existing Cucumber tools.

List of Figures

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

3.1 AST for top-level properties example 16
3.2 AST for a list access example 17
3.3 AST for a `v-for` substitution example 17
3.4 Example Graph obtained for the identifier `this.problems[0].a` 19
3.5 Example Graph for the following object accesses - `this.problem.a`, `this.problem.b`, `this.problem.author.name` 19
3.6 Abstract list representation 20
3.7 Concrete example of a list representation 20
3.8 Example Interaction Diagram graph including list elements with properties. Circled nodes represent data node and rectangular ones HTML tags and methods. 22

5.2 Math for Kids in Vue.js generated interaction diagram 34
5.3 Math for Kids in AngularJS interaction diagram by Zhang and Zhao [ZZ19] 35
5.4 Gherkin scenario templates and sample human written scenarios based on the templates 36
5.5 Math for Kids in Vue.js including subtraction problems, more precise statistics and display of past problems 38
5.6 Generated interaction diagram for Math for Kids including subtraction in Vue.js . 39
5.7 The menu with daily meal application 39
5.8 Generated interaction diagram for Menu with daily meal 41

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.

[Vue21a] Vuejs. *Vue.js*. [Online; accessed 29. Jan. 2021]. Jan. 2021. URL: https://github.com/vuejs/vue.

[And+17] Esben Andreasen et al. "A survey of dynamic analysis and test generation for JavaScript". In: *ACM Computing Surveys (CSUR)* 50.5 (2017), pp. 1–36.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: Directed automated random testing". In: *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 2005, pp. 213–223.

[cra21] crawljax. *Crawljax: Crawling Dynamic (JavaScript-based) Web Applications*. [Online; accessed 10. Feb. 2021]. Feb. 2021. URL: https://github.com/crawljax/crawljax.

[Gao+19] Pengfei Gao et al. "Model-based Automated Testing of JavaScript Web Applications via Longer Test Sequences". In: *arXiv preprint arXiv:1905.07671* (2019).

[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.

[21a] *Who Does What? - Cucumber Documentation*. [Online; accessed 10. Feb. 2021]. Feb. 2021. URL: https://cucumber.io/docs/bdd/who-does-what.

[21b] *Gherkin Reference - Cucumber Documentation*. [Online; accessed 10. Feb. 2021]. Feb. 2021. URL: https://cucumber.io/docs/gherkin/reference.

[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.

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

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

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

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

[est21] 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.

[EST21] ESTree. *ESTree.* [Online; accessed 27. Jan. 2021]. Jan. 2021. URL: https://github.com/estree/estree.

[Vue21c] Vuejs. *vue-eslint-parser AST.* [Online; accessed 27. Jan. 2021]. Jan. 2021. URL: https://github.com/vuejs/vue-eslint-parser/blob/master/docs/ast.md.

[21f] *Typed JavaScript at Any Scale.* [Online; accessed 11. Feb. 2021]. Feb. 2021. URL: https://www.typescriptlang.org.

[21g] *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.

[21h] *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.

[Kar21] Anton Karakochev. *vuejs-example*. [Online; accessed 10. Feb. 2021]. Feb. 2021. URL: https://github.com/KarakoA/vuejs-example.

[MMP15] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. "JSeft: Automated JavaScript unit test generation". In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2015, pp. 1–10.

[Zha20] Gefei Zhang. "Specifying and Model Checking Workflows of Single Page Applications with TLA+". In: *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE. 2020, pp. 406–410.

Appendix A

Appendix

A.1 Submitted Source Code

Together with this thesis a *.zip* file is submitted, which contains the source code of the application.

A.2 ESLint

A.2.1 Notable AST Nodes

ObjectExpression

```
interface ObjectExpression <: Expression {
    type: "ObjectExpression";
    properties: [ Property ];
}
```

Property

```
interface Property <: Node {
    type: "Property";
    key: Literal | Identifier;
    value: Expression;
    kind: "init" | "get" | "set";
}
```

VElement

```
interface VElement <: Node {
  type: "VElement"
  namespace: string
  name: string
  startTag: VStartTag
  children: [ VText | VExpressionContainer | VElement ]
```

```
  endTag: VEndTag | null
  variables: [ Variable ]
}
```

VText

```
interface VText <: Node {
  type: "VText"
  value: string
}
```

VAttribute

```
interface VAttribute <: Node {
  type: "VAttribute"
  directive: false
  key: VIdentifier
  value: VLiteral | null
}
```

Identifier

```
interface Identifier <: Expression, Pattern {
  type: "Identifier";
  name: string;
}
```

MemberExpression

```
interface MemberExpression <: Expression, Pattern {
  type: "MemberExpression";
  object: Expression;
  property: Expression;
  computed: boolean;
}
```

CallExpression

```
interface CallExpression <: Expression {
  type: "CallExpression";
  callee: Expression;
  arguments: [ Expression ];
}
```

AssignmentExpression

```
extend interface AssignmentExpression {
    left: Pattern;
}
```

A.3 Source Code

A.3.1 Math for Kids

```
<template>
    <div>
      <div style="margin-left: 5px; margin-top: 20px">
        <form @submit.prevent>
          <span>{{ a }}</span> + <span> {{ b }}</span> =
          <span><input class="question" v-model="answer" /></span>
          <button
            class="btn btn-primary"
            style="margin-left: 5px"
            :disabled="!may_check()"
            type="submit"
            @click="check_answer()"
          >
            Check
          </button>
          <button
            class="btn btn-success"
            style="margin-left: 5px"
            disabled="1"
            v-if="right === true"
          >
            Right
          </button>
          <button
            class="btn btn-danger"
            style="margin-left: 5px"
            disabled="1"
            v-if="right === false"
          >
            Wrong
          </button>
          <button
            class="btn btn-info"
            style="margin-left: 5px"
            @click="new_problem()"
          >
            New Problem
          </button>
        </form>
      </div>
```

```
    <hr />
    <div style="margin-left: 5px">
      <table>
        <tr>
          <td><strong>Statistics </strong></td>
          <td><span class="badge badge-success">Right</span></td>
          <td><span class="badge badge-danger">Wrong</span></td>
        </tr>
        <tr>
          <td />
          <td>
            <span> {{ count_right }}</span>
          </td>
          <td>
            <span> {{ count_wrong }}</span>
          </td>
        </tr>
      </table>
    </div>
  </div>
</template>

<script>
export default {
  name: "HelloWorld",
  props: {
    msg: String,
  },
  data() {
    return {
      a: 0,
      b: 0,
      c: 0,
      answer: undefined,
      right: undefined,
      count_right: 0,
      count_wrong: 0,
    };
  },
  created: function () {
    this.add_problem();
  },
  methods: {
    add_problem() {
      let max = 100;
      this.c = Math.floor(Math.random() * (max - 1)) + 1;
      this.a = Math.floor(Math.random() * (this.c - 2)) + 1;
      this.b = this.c - this.a;

      this.answer = undefined;
      this.right = undefined;
    },
```

```
      check_answer() {
        this.right = this.c === parseInt(this.answer);
        this.right ? (this.count_right += 1) : (this.count_wrong += 1);
      },
      may_check() {
        // answer non-empty and right undefined
        return (
          !(this.answer === undefined || this.answer === "") &&
          this.right === undefined
        );
      },
      new_problem() {
        this.add_problem();
      },
    },
  };
  </script>
  <style scoped>
  .index {
    padding: 0 20px 0 5px;
  }

  .question {
    font-color: white;
    width: 2em;
    background-color: #3ff;
    margin-left: 5px;
    margin-right: 5px;
  }
  </style>
```

A.3.2 Math for Kids Extended

```
<template>
    <div>
      <ul>
        <li v-for="problem in problems" :key="problem.id">
          <div style="margin-left: 5px; margin-top: 20px">
            <form @submit.prevent>
              <span>{{ problem.a }}</span>
              <span v-if="problem.isAddProblem"> + </span>
              <span v-if="!problem.isAddProblem"> - </span>
              <span> {{ problem.b }}</span> =
              <span><input class="question" v-model="problem.answer" /></span>
               <button
                class="btn btn-primary"
                style="margin-left: 5px"
                :disabled="!may_check(problem)"
                type="submit"
                @click="check_answer_add(problem)"
                v-if="problem.isAddProblem"
```

```
            >
              Check
            </button>
             <button
              class="btn btn-primary"
              style="margin-left: 5px"
              :disabled="!may_check(problem)"
              type="submit"
              v-if="!problem.isAddProblem"
              @click="check_answer_sub(problem)"
            >
              Check
            </button>
            <button
              class="btn btn-success"
              style="margin-left: 5px"
              disabled="1"
              v-if="problem.right === true"
            >
              Right
            </button>
            <button
              class="btn btn-danger"
              style="margin-left: 5px"
              disabled="1"
              v-if="problem.right === false"
            >
              Wrong
            </button>
            <button
              class="btn btn-info"
              style="margin-left: 5px"
              @click="new_problem()"
            >
              New Problem
            </button>
          </form>
        </div>
      </li>
    </ul>
    <hr />
    <div style="margin-left: 5px">
      <table>
        <tr>
          <td><strong>Statistics </strong></td>
          <td><span class="badge badge-success">Right</span></td>
          <td><span class="badge badge-danger">Wrong</span></td>
          <td><span class="badge badge-info">Accuracy</span></td>
        </tr>
        <tr>
          <td>
            <span> Addition </span>
```

```
          </td>
          <td>
            <span> {{ count_right_add }}</span>
          </td>
          <td>
            <span> {{ count_wrong_add }}</span>
          </td>
          <td>
            <span> {{ accuracy_add }}</span>
          </td>
        </tr>

        <tr>
          <td>
            <span> Subtraction </span>
          </td>
          <td>
            <span> {{ count_right_sub }}</span>
          </td>
          <td>
            <span> {{ count_wrong_sub }}</span>
          </td>
          <td>
            <span> {{ accuracy_sub }}</span>
          </td>
        </tr>
      </table>
    </div>
  </div>
</template>

<script>
export default {
  name: "HelloWorld",
  props: {
    msg: String,
  },
  data() {
    return {
      problems: [],
      count_right_add: 0,
      count_wrong_add: 0,
      count_right_sub: 0,
      count_wrong_sub: 0,
    };
  },
  computed: {
    accuracy_add: function () {
      let n = this.count_wrong_add + this.count_right_add;
      return n == 0 ? 0 : this.count_right_add / n;
    },
    accuracy_sub: function () {
```

```
      let n = this.count_wrong_sub + this.count_right_sub;
      return n == 0 ? 0 : this.count_right_sub / n;
    },
  },
  created: function () {
    this.add_problem();
  },
  methods: {
    add_problem() {
      let max = 100;

      let isAddProblem = Math.random() > 0.5;

      let c1 = Math.floor(Math.random() * (max - 1)) + 1;
      let a1 = Math.floor(Math.random() * (c1 - 2)) + 1;
      let b1 = c1 - a1;
      let id = this.problems.length + 1;

      let problem = isAddProblem
        ? {
            c: c1,
            a: a1,
            b: b1,
            id,
            answer: undefined,
            right: undefined,
            isAddProblem,
          }
        : {
            c: b1,
            a: c1,
            b: a1,
            id,
            answer: undefined,
            right: undefined,
            isAddProblem,
          };
      this.problems.push(problem);
    },

    check_answer_sub(problem) {
      problem.right = problem.c === parseInt(problem.answer);
        problem.right
          ? (this.count_right_sub += 1)
          : (this.count_wrong_sub += 1);
    },
    check_answer_add(problem) {
      problem.right = problem.c === parseInt(problem.answer);
        problem.right
          ? (this.count_right_add += 1)
          : (this.count_wrong_add += 1);
    },
```

```
      may_check(problem) {
        // answer non-empty and right undefined
        return (
          !(problem.answer === undefined || problem.answer === "") &&
          problem.right === undefined
        );
      },
      new_problem() {
        this.add_problem();
      },
    },
  };
  </script>
  <style scoped>
  .index {
    padding: 0 20px 0 5px;
  }

  .question {
    font-color: white;
    width: 2em;
    background-color: #3ff;
    margin-left: 5px;
    margin-right: 5px;
  }
  </style>
```

A.3.3 Menu with daily meal

```
<template>
    <div style="margin-left: 20px">
    <div>Menu</div>
      <ul>
        <li v-for="meal in meals" :key="meal.id">
          <div style="margin-left: 5px; margin-top: 20px">
            <div> {{meal.name}}</div>
            <div>Price {{meal.price}} Euro </div>
          </div>
        </li>
      </ul>
      <div>
  </div>
  <div>Meal of the day - {{mealOfTheDay.name}}</div>
  <div style="margin-top: 20px"/>
  <div>Best steak in town - {{meals[2].price}} Euro !</div>
  <div style="margin-top: 20px"/>
  <label>Store manager commands:</label>
  <div>
  <button v-on:click="new_day">New Day</button>
  <button v-on:click="discount_all">Discount all</button>
  </div>
```

```
</div>
</template>

<script>
export default {
  name: "HelloWorld",
  data() {
    return {
      answers: [],
      mealOfTheDay: undefined,
      isFriday: false
    };
  },
  created: function () {
    this.init();
  },
  methods: {
    init() {
      this.meals = [
        {name:"Soup",  id:0, price: 1 },
        {name:"Salad", id:1, price: 2 },
        {name:"Steak", id:2, price: 7 },
      ]
      this.new_day();
    },
    new_day:function(){
      this.isFriday = !this.isFriday
      if(this.isFriday){
        this.meals[0].price = this.meals[0].price * 2

        this.meals[1].price = this.meals[1].price / 2
        this.mealOfTheDay= this.meals[1]
      }
      else {
        this.meals[1].price = this.meals[1].price * 2

        this.meals[0].price = this.meals[0].price / 2
        this.mealOfTheDay = this.meals[0]
      }
    },
    discount_all:function(){
    for (var i = 0; i < this.meals.length; i++) {
      this.meals[i].price = this.meals[i].price - 0.20
    }
    }
  },
};
</script>
```

A.4 Generated Gherkin Scenarios

A.4.1 Math for Kids Extended

```
Scenario: ['created']
        When 'created'
        Then 'problems[i].a'
        And '+'
        And '-'
        And 'Check'
        And 'Check'
        And 'problems[i].b'
        And 'problems[i].answer'
        And 'Right'
        And 'Wrong'

Scenario: ['created', 'problems[i].answer']
        Given 'created'
        When 'problems[i].answer'
        Then 'problems[i].answer'
        And 'Check'
        And 'Check'

Scenario: ['created', 'Check']
        Given 'created'
        When 'Check'
        Then 'Right'
        And 'Wrong'
        And 'Check'
        And 'Check'
        And 'count_right_add'
        And 'accuracy_add'
        And 'count_wrong_add'

Scenario: ['created', 'Check']
        Given 'created'
        When 'Check'
        Then 'Right'
        And 'Wrong'
        And 'Check'
        And 'Check'
        And 'count_right_sub'
        And 'accuracy_sub'
        And 'count_wrong_sub'

Scenario: ['created', 'New Problem']
        Given 'created'
        When 'New Problem'
        Then 'problems[i].a'
        And '+'
        And '-'
        And 'Check'
```

```
        And 'Check'
        And 'problems[i].b'
        And 'problems[i].answer'
        And 'Right'
        And 'Wrong'

Scenario: ['created', 'problems[i].answer', 'Check']
        Given 'created'
        And 'problems[i].answer'
        When 'Check'
        Then 'Right'
        And 'Wrong'
        And 'Check'
        And 'Check'
        And 'count_right_add'
        And 'accuracy_add'
        And 'count_wrong_add'

Scenario: ['created', 'Check', 'Check']
        Given 'created'
        And 'Check'
        When 'Check'
        Then 'Right'
        And 'Wrong'
        And 'Check'
        And 'Check'
        And 'count_right_add'
        And 'accuracy_add'
        And 'count_wrong_add'

Scenario: ['created', 'New Problem', 'Check']
        Given 'created'
        And 'New Problem'
        When 'Check'
        Then 'Right'
        And 'Wrong'
        And 'Check'
        And 'Check'
        And 'count_right_add'
        And 'accuracy_add'
        And 'count_wrong_add'

Scenario: ['created', 'New Problem', 'problems[i].answer']
        Given 'created'
        And 'New Problem'
        When 'problems[i].answer'
        Then 'problems[i].answer'
        And 'Check'
        And 'Check'

Scenario: ['created', 'New Problem', 'problems[i].answer', 'Check']
        Given 'created'
```

```
        And 'New Problem'
        And 'problems[i].answer'
        When 'Check'
        Then 'Right'
        And 'Wrong'
        And 'Check'
        And 'Check'
        And 'count_right_add'
        And 'accuracy_add'
        And 'count_wrong_add'
```