Visualizing Interactions in AngularJS-based Single Page Web Applications*

Gefei Zhang†
Hochschule für Technik und Wirtschaft Berlin
gefei.zhang@htw-berlin.de

Jianjun Zhao
Kyushu University
zhao@ait.kyushu-u.ac.jp

Abstract

AngularJS is a popular framework for single page web applications. In AngularJS applications, the programming logic is implemented in Javascript, while the layout is defined separately in HTML files. Due to this separation, data and control flow is usually hard to track. We propose a method to visualize the data and control flow in AngularJS-based single page web applications and separate interactions from each other. Our method helps to get a better understanding of the application’s work flow, to realize the boundaries of the interactions, and to know what is updated in an interaction and what is not.

1 Introduction

AngularJS [2] is one of the modern frontend-frameworks which support the Model-View-ViewModel architecture (MVVM) [8]: the models provide data to the application, the views define the graphical presentation of the data, and the view-models (also called controllers) define the business logic (data and control flows) of the application. Usually, views are defined in HTML, models and controllers in Javascript.

AngularJS is widely used in single page web applications (SPA). In an SPA, the application has only one HTML page, containing an array of widgets. When the user gives some input in one widget, the application reacts and updates some other widgets. Between the widgets of the page there may or may not exist data and control flow, and a widget may or may not be influenced by another one. Since data and control flow is defined in the controller, separately from the widgets, potential interactions may be obscure; understanding of the program may be hard.

We present a method to visualize data and control flow of AngularJS-based SPA. We create an Interaction Diagram by translating HTML widgets, as well as functions and variables in the controller to nodes, and the invocation, reading and writing relationships between them as edges. The interaction diagram not only visualizes possible workflows of the application, but is also a staring point for more static analysis. In this paper, we show how to calculate “slices” of interactions, that is, to isolate the widgets involved in an interaction from those that are not.

The rest of this paper is organized as follows: In the following Sect. 2, we give a brief introduction to AngularJS, and also present our running example. Section 3 introduces interaction diagrams. In Sect. 4 we show how to analyse workflows of the application and define test cases using the interaction diagram. Related work is discussed in Sect. 5. Finally, in Sect. 6, we conclude and outline some future work.

2 AngularJS

We first give a brief introduction to AngularJS by means of a simple example, and then define an abstract syntax for AngularJS applications. Due to space limitation, we focus on a small subset of AngularJS; it is relatively straight-forward to extend our approach to cover other features of the framework.

2.1 Running Example

Figure 1 shows an AngularJS application to teach kids addition. In the upper part an addition problem is presented, the lower part shows statistics of how many right and wrong answers the user has given. When a new problem is shown, the user can enter her answer in an input field (Fig. 1(a)), the system then shows if the answer is right or wrong. Meanwhile, a new button appears. When the user clicks on it, a new problem is generated (Fig. 1(b)). In the lower part of the browser, a statistics is shown of how many right and wrong answers the user has given. In Fig. 1(b), the user has answered four questions in total, where three of the answers were correct, and one was wrong.

2.2 Data binding

In AngularJS, the program logic is implemented in a so-called controller in Javascript, and the graphical
AngularJS supports both one-way and two-way data binding. The directive `ng-bind` or double braces `{{}}` define one-way data binding, that is, the HTML element automatically presents the up-to-date value of the bound property of `$scope`, whilst changes of the value presented by the HTML element, if any, would not be propagated from the GUI back to the controller (the `$scope`). For instance, in Fig. 2, line 5, `{{}}` will be replaced by the values of the variables `$scope.a` and `$scope.b` at runtime, respectively. The values of the two variables are assigned in function `add_problem` of the controller (Fig. 3, lines 11 and 12). A little more examination of the controller reveals that this function is called to generate a new addition problem, and `$scope.a` and `$scope.b` hold the values of the two summands.

Other directives defining one-way data binding include `ng-if` and `ng-disabled`. In `ng-if`, the value of the bound property is not shown; instead it regulates the visibility of the HTML element: if and only if the Javascript expression bound to `ng-if` is valued to `true` is the HTML element visible in the GUI. For example, both of the two buttons defined in lines 8 and 9 of Fig. 2 are guarded with `ng-if`. The conditions for the buttons being visible are the variable `$scope.right` being `true` and `false`, respectively. Figure 3 shows that the variable is set in line 22, in function `$scope.check_answer`, and is `true` iff the value of variable `c` equals to the value of `$scope.answer` parsed as an integer.

Directive `ng-disabled` is used to set HTML widgets disabled (i.e., the user cannot enter her input). In line 6 of Fig. 2, the button is disabled when the negation of the result of function `$scope.may_check()`, which is defined in the lines 17–19 of Fig. 3. The function returns `true` iff the input field is not empty (`!!$scope.answer`) and `$scope.right` is still `undefined` (i.e., the button `Check` has not been clicked yet, see Sect. 2.3). When this is the case the HTML code

---

1The whole project is available under https://bitbucket.org/gefei/angularjs-example

2There is some subtle difference between the semantics of `ng-bind` and double braces, see https://stackoverflow.com/a/16126174. However, the difference is not relevant for our discussion. In this paper, we consider `ng-bind` and double braces as equivalent.
var c;

function add_problem() {
  var max = 100;
  c = Math.floor(Math.random() * (max - 1)) + 1;
  $scope.a = Math.floor(Math.random() * (c - 2)) + 1;
  $scope.b = c - $scope.a;
  $scope.answer = undefined;
  $scope.right = undefined;
}

$scope.count_right += ($scope.right) ? 1 : 0;
$scope.right = c === parseInt($scope.answer);

$scope.may_check = function() {
  return !$scope.answer && $scope.right === undefined;
}

$scope.check_answer = function() {
  $scope.right = c === parseInt($scope.answer);
  $scope.count_right += ($scope.right) ? 1 : 0;
  $scope.count_wrong += ($scope.right) ? 0 : 1;
}

$scope.new_problem = function() {
  add_problem();
}

add_problem();

Figure 2. HTML template

We call the property of $scope that an HTML element is bound to the target of the data binding, and the HTML element the source. While ng-bind and {{}} show the value of the target in the GUI, ng-if and ng-disabled do not output the value in the textual form. It rather influences the appearance of the source by setting its visibility or enabled/disabled status. In both cases, we say the source of a data binding presents the value of the target.

The directive ng-model defines two-way data binding, that is, the widget automatically updates when the value of its bound property changes, and any change of the widget’s value will be propagated to the bound variable. Therefore, a bi-directional data flow is defined. For instance, in line 5 of Fig. 2, the input element has an attribute ng-model="answer". The input field is therefore bound to the variable $scope.answer: the value of the input field is hold in $scope.answer, changes are propagated automatically in both directions. In Fig. 3, the correct answer of the current problem is stored in variable c (line 10), and $scope.right is calculated by a comparison with $scope.answer in line 22.

The target of a one-way data binding may be a variable property or a function property of $scope. If it is a variable property, the source presents the target’s value. If it is a function property, the widget presents the return value of the function. On the other hand, the target of a two-way data binding must be a variable property, since the target must store the value of the user input, and only a variable, as opposed to a function, can be assigned a value.

The template of the statistics is defined in an HTML table (lines 14 to 25 in Fig. 2), where two td cells (lines 22 and 23) present the values of $scope.count_right and $scope.count_wrong by one-way data binding.

2.3 Event Handling

HTML elements may also be provided with event handlers. Upon the given event, the specified function is executed. For example, in line 7 of Fig. 2, the directive ng-click binds the function $scope.check_answer to the button. Therefore, the function is invoked when the user clicks the button. The function is defined in Fig. 3, lines 21 to 25. When invoked, it first (line 22) checks if c has the same value as $scope.answer (which we know is the current value of the user input in the text field, see Sect. 2.2), parsed as an integer, and assigns the result to $scope.right. Then, depending on if $scope.right is true or not, the value $scope.count_right or $scope.count_wrong is incremented by one. Therefore, the statistics gets updated when the user clicks the Check button (Fig. 2, lines 22 and 23).

The button New Problem also has an event handler: in Fig. 2, line 10, the directive ng-click="new_problem()" binds the function $scope.new_problem to handle the event of the button being clicked. The function calls another function add_problem to generate a new problem by updating the values of $scope.a, $scope.b, setting c, to be undefined to clear the field for the user to in-
put her answer (recall: this field has a two-way data binding, as defined in line 5 of Fig. 2), setting both $\text{scope.right}$ and $\text{scope.answer}$ to be undefined. Therefore, when \text{New Problem} is clicked, $\text{scope.may_check}$ will return true, and the button Check will get enabled (Fig. 2, line 6).

Variable $c$ and function \text{new_problem} are not defined as $\text{scope}$’s properties. Therefore, they are local to the controller, and not exposed to the template.

2.4 Abstract Syntax

An AngularJS-based SPA is a tuple $(T, C, D, E)$. $T$ is a template, written in HTML and consisting of a set of HTML tags $(T = \{h\})$ which define HTML widgets,$^3$ $C$ is the definition of a controller, written in Javascript, $D$ is a set of data bindings, and $E$ is a set of event handler bindings.

The controller definition $C$ is modeled as a tuple $(V, F, \text{scope})$, where $V$ is a set of top-level variables, $F$ is a set of top-level functions, and $\text{scope} \in V$ is a distinguished element of $V$. We write $V(\text{scope})$ for the set of $\text{scope}$’s variable properties, and $F(\text{scope})$ for the set of $\text{scope}$’s function properties. We also define $W = V \setminus \{\text{scope}\}$ to be the set of top-level variables defined in the controller other than $\text{scope}$.

$D$ is the set of data binding relations between HTML tags and variable properties of $\text{scope}$: $D \subseteq \{(h, V(\text{scope}) \cup F(\text{scope}))\}$. Given $d = (n, o) \in D$, we define $\text{source}(d) = n$ and $\text{target}(d) = o$. Two-way data bindings build a subset $D' \subseteq D$, and $\forall d \in D'$, it holds that $\text{target}(d) \in V(\text{scope})$.

$E$ is the set of event handler bindings between HTML tags and function properties of $\text{scope}$: $E \subseteq \{(h, F(\text{scope}))\}$.

Additionally, for each $f \in F \cup F(\text{scope})$, we define $R(f) \subseteq V \cup V(\text{scope})$ and $W(f) \subseteq V \cup V(\text{scope})$ to be the set of the variables $f$ reads from and writes to, respectively. We also define $\text{Inv}(f) \subseteq F$ to be the functions invoked by $f$.\(^4\) In this paper, we take these sets for granted. Since Javascript is an “extremely dynamic” \cite{7} language, this is in general not always the case. However, in modern software development, \textit{readability first} is considered best practice, and it is reasonable to assume that at least reading, writing, and invocation relationships can be obtained by simple analysis.

3 Interaction Diagram

In order to understand the workflow of the application, it is necessary to study both its HTML and its Javascript code, as well as their interactions. We now present Interaction Diagrams to visualize the overall behavior, combining the logic defined in HTML and Javascript. Figure 4 shows the interaction diagram for our running example.

An Interaction Diagram $ID$ is a directed graph $(N, E)$. The set of nodes is defined as the union of three sets: $N = N_T \cup N_{\text{scope}} \cup N_{\text{js}}$, where, using the notations introduced in Sect.2.4,

- for each $(h, v) \in D$, we generate a node $n_h$, and $N_T = \{n_h \mid (h, v) \in D\}$. Graphically, we label $n_h$ with the label of $h$, or, if $h$ does not have a label, the name of the target of the data binding (without $\text{scope}$), extended by the position of the definition of $h$ in the template;
- for each $(h, v) \in D$, we create a node $n_v$, and for each $(h, e) \in E$, we create a node $n_e$. $N_{\text{scope}}$ is then defined as $N_{\text{scope}} = \{n_v \mid (h, v) \in D\} \cup \{n_e \mid (h, e) \in E\}$. Graphically, we label each $n_e \in N_{\text{scope}}$ with the name of the $f$ (with $\text{scope}$), but do not include the position of $f$’s appearances in the controller.
- for each $v \in W$, we create a node $n_v$; for each $f \in F$, we create a node $n_f$, and $N_{\text{js}} = \{n_v \mid v \in W\} \cup \{n_f \mid f \in F\}$. Graphically, we label these nodes with the name of the variable or function.

For example, Fig. 4 shows the interaction diagram for our running example. The upper compartment shows $N_T$, which contains a node for each of the text fields $a$, $b$ (which show the two summands to the user), $\text{scope.answer}$ (where the user may enter her answer), the buttons Check (which the user may click to check if the answer is correct), the labels Right and Wrong (which show the result of the check to the user), the labels count_right and count_wrong (which show the current success statistics of the user), and the button New Problem (which the user may click to generate a new problem).

The lower compartment models components of the controller. It contains the elements of $N_{\text{scope}}$ and $N_{\text{js}}$. That is, it contains a node for each of the variables $\text{scope.a}$, $\text{scope.b}$ (which hold the values of the summands), $\text{scope.answer}$ (which holds the answer inputted by the user), $\text{scope.right}$ (which holds the value $\text{scope.check_answer}$ returns, see below), $\text{scope.count_right}$ and $\text{scope.count_wrong}$ (which hold the current number of right and wrong answers the user has given), and the functions $\text{scope.check_answer}$ (which checks if the answer inputted by the user is correct), and $\text{scope.new_problem}$ (which generates the summands and key of a new problem). The lower compartment also contains a node for $c$, which holds the key of the new problem.

The edges $E$ of the interaction diagram are used to model data and control flow. We define $E$ as the union of six subsets: $E = E_{\text{data}} \cup E_{\text{data}} \cup E_{\text{event}} \cup E_{\text{js}} \cup E_{\text{R}} \cup E_{\text{Inv}}$, where

\(^3\)Precisely, HTML tags may be nested, thus the HTML template is a tree. In this paper, we do not consider nesting tags, and just view the template as a set.

\(^4\)Although it is also possible for a $\text{scope}$ function to call another $\text{scope}$ function, it is usually not necessary and not a good idea.
4 Interactions between Widgets

SPAs are interactive: the user makes some input, the system reacts and makes updates to some widgets, then the user makes another input, and so on. We define an interaction to be a round of user giving input, and the system updating widgets accordingly. An interaction can be triggered explicitly by the user invoking an event handler, or implicitly while the user is updating data, which is bound by ng-model. Starting from interaction diagrams, it is easy to “slice” interactions, i.e., to find out the widgets that get updated upon a certain piece of user input.

A widget $t$ reacts to another widget $s$ iff in the interaction diagram $t$’s representation $n_t$ is reachable from $s$’s presentation $n_s$, and the only event-handling edge, if any, on the path from $n_s$ to $n_t$ event-handling edge is the very first edge (leaving $n_s$) on the path. This edge models the interaction being explicitly triggered.

Formally, given a node $n \in \mathbb{N}_H$, we say a node $m \in \mathbb{N}_H$ reacts to $n$ iff

1. $\exists n_0, n_1, n_2, \ldots, n_k \in \mathbb{N}, n_0 = n, n_k = m$ 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 $e \notin E_{event}$.

For each $d \in D$, we create an edge $e_d = (\text{target}(d), \text{source}(d))$. The set of all data-flow edges from the controller to HTML widgets is then modeled as $E_{data} = \{e_d \mid d \in D\}$. If $d \in D'$, we additionally create an edge $e'_d = (\text{source}(t), \text{target}(t))$ to model the HTML widget reading value from its bound variable. The set of all data-flow edges from HTML widgets to the controller is then modeled as $E'_{data} = \{e_d \mid d \in D'\}$.

For each $(h, f) \in E$, we define an event-handling edge $e_h = (n_h, n_f)$. The set of all event-handling control flow is then modeled as $E_{event} = \{e_h \mid (h, f) \in E\}$.

For each pair $(f, v), f \in F \cup F(\$scope), v \in W(f)$, we create an edge $e_{f,v}$. The set of writing relations is then modeled as $E_W = \bigcup_{e \in F \cup F(\$scope)} \{e_{f,v} \mid v \in W(f)\}$. For each pair $(v, f), f \in F \cup F(\$scope), v \in R(f)$, we create an edge $e_{v,f}$. The set of reading relations is then modeled as $E_R = \bigcup_{e \in F \cup F(\$scope)} \{e_{v,f} \mid v \in R(f)\}$.

For each $f \in F \cup F(\$scope)$ and each $v \in \text{lnv}(f)$, we create an edge $e_{f,v}$. The relations of a function writing a variable is then modeled by $E_{inv} = \bigcup_{e \in F \cup F(\$scope)} \{e_{v,f} \mid v \in \text{lnv}(f)\}$.

Graphically, we use a dashed-line arrow to represent each $e \in E_{data} \cup E'_{data} \cup E_W \cup E_R \cup E_{inv}$, and a solid-line arrow to represent each $e \in E_{event}$. Note for event handling, the exact event is not modeled, and it does not need to be modeled in an Interaction Diagram.

In our example (see Fig. 4), the elements of $E_{data}$ model one-way data binding: $(\$scope.a, a), (\$scope.b, b), (\$scope.count_right, count_right), (\$scope.CountWrong, count_wrong), (\$scope.right, Right), (\$scope.Wrong, Wrong)$, and $(\$scope.may_check(), Check)$. $E_{data}$ contains the two edges between $\$scope.answer and answer, modeling the only two-way data binding in the example. $E_{event}$ contains the two edges (Check, $\$scope.check_answer()) and (New Problem, $\$scope.new_problem()). Furthermore, $E_W$ contains the four edges leaving add_problem(), and the edges leaving $\$scope.check_answer(). $E_R$ contains the edges from c to $\$scope.check_answer(), and from $\$scope.answer to $\$scope.may_check(). $E_{inv}$ contains the edge from $\$scope.new_problem() to add_problem().
We write \( I(n) \) 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. To analyze interactions in the SPA, we calculate for each input widget, i.e., widget with an edge leaving it in the interaction diagram. Based on Fig. 4, we can calculate the following interactions for our running example:

- \( I(\text{answer}) = \{ \text{Check, answer} \} \), which means when the user is entering her answer, \text{answer} (which is trivial) and \text{Check} (enabled) are updated. Note that updating the answer does not trigger \( \$\text{scope.check\_answer}() \), since this function needs explicit triggering via \text{Check}.
- \( I(\text{Check}) = \{ \text{Check, Right, Wrong, count\_right, count\_wrong} \} \), which means when the user clicks on \text{Check}, these widgets get an update: the button itself (disabled), one of \text{Right} and \text{Wrong} is displayed, indicating whether the user-inputted answer is correct, and one of the counts also gets updated.
- \( I(\text{New Problem}) = \{ a, b, \text{answer}, \text{Check, Right, Wrong} \} \), which means when the user clicks \text{New Problem}, the widgets \text{a}, \text{b} (showing the summands of the new problem), \text{answer} (emptied), \text{Check} (enabled), as well as \text{Right} and \text{Wrong} (both made invisible), are updated.

This analysis shows us the boundary of the interactions. For instance, according to the analysis, when the user is updating an answer, the label \text{Right} or \text{Wrong} is not shown, nor does the statistics get updated. Instead, these widgets are only updated when the user clicks \text{Check}.

5 Related Work

Analysis of Javascript programs is a very dynamic research field. Due to the very dynamic nature of the language, its analysis is not an easy task, see [7] for an overview of recent publications. One of the challenges is the interactions between browser, DOM and Javascript. In [6, 3], a unified API is given to formalise the browser behavior. However, large-scale libraries are still difficult to analyze, and therefore their functionalities are often modeled manually. Our work is also along this line in that we also take the semantics of AngularJS for granted.

For the analysis of such frameworks, it is essential to understand their interactions [7]. This is exactly where our work is positioned. Other interesting publications in this area include [4], which provides a method of checking name and type consistency between template and controller, and [1], where the authors present a hybrid method for change impact analysis, and its focused on plain Javascript, without frameworks. Compared with these approaches, our focus is on the visualization and analysis of boundaries of interactions in applications based on a complex framework.

6 Conclusions and Future Work

We presented a method for visualization and analyzing AngularJS-based single page web applications. Based on the interaction diagrams, it is easy to calculate the interactions, and to understand which widgets react to certain user input and which do not. Our approach is helpful for understanding AngularJS programs, and thus for more powerful analysis.

In the future, we plan to automate this approach, to extend the analysis by more AngularJS directives, and based on this work, to investigate techniques for automatic test generation for AngularJS programs.

References