Detecting Inconsistencies in JavaScript MVC Applications

Frolin S. Ocariza, Jr. Karthik Pattabiraman Ali Mesbah
University of British Columbia
Vancouver, BC, Canada
{*frolino, karthikp, amesbah*}*@ece.ubc.ca*

***Abstract*—Higher demands for more reliable and maintainable JavaScript-based web applications have led to the recent development of MVC (Model-View-Controller) frameworks. One of the main advantages of using these frameworks is that they abstract out DOM API method calls, which are one of the leading causes of web application faults, due to their often complicated interaction patterns. However, MVC frameworks are susceptible to inconsistencies between the identifiers and types of variables and functions used throughout the application. In response to this problem, we introduce a formal consistency model for web applications made using MVC frameworks. We propose an approach – called AUREBESH – that automatically detects inconsistencies in such applications. We evaluate AUREBESH by conducting a fault injection experiment and by running it on real applications. Our results show that AUREBESH is accurate, with an overall recall of 96.1% and a precision of 100%. It is also useful in detecting bugs, allowing us to find 15 real-world bugs in applications built on AngularJS, a popular MVC framework.**

I. INTRODUCTION

With the usage of client-side JavaScript in web applications becoming more and more ubiquitous, there is increasing demand to write JavaScript code that is reliable and maintainable. In an empirical study [1], we analyzed over 300 bug reports to learn what specific characteristics JavaScript faults possess (i.e., their root cause, impact, and propagation). We found that the majority of reported JavaScript faults are *DOM-related*, meaning the error leading to the fault propagates into the parameter value of a DOM API method call.[1] Such faults often result from a developer's incomplete or erroneous understanding of the relationship between the JavaScript code and the DOM, leading to inconsistent interactions between these two entities, thereby causing failures.

Partly in response to these issues, JavaScript libraries known as *MVC frameworks* have recently been developed. MVC frameworks such as AngularJS [2], BackboneJS [3], and Ember.js [4] use the well-known Model-View-Controller (MVC) pattern to simplify JavaScript development in a way that abstracts out DOM method calls. This is accomplished by giving programmers the ability to define model objects, which are then directly embedded in the HTML code (typically via a double curly brace notation) such that any changes in these objects' values will automatically be reflected in the DOM, and vice versa – a process known as "two-way data binding". The frameworks thus eliminate the need for web programmers to explicitly set up DOM interactions in JavaScript.

Unfortunately, despite the apparent advantages, MVC frameworks are still susceptible to consistency issues akin to DOM-JavaScript interactions [5]. In particular, these frameworks rely on the use of *identifiers* to represent model objects and controller methods; definitions and uses of these identifiers are expected to be consistent across associated models, views, and controllers. Moreover, due to JavaScript's loose typing, which is retained in these MVC frameworks, the programmer must ensure that the values assigned to model objects and returned by controller methods are consistent with their expected *types*, depending on how they are used. Since model objects and controller methods are primarily used to represent major functionalities of the web application, any inconsistencies between these identifiers and types can potentially lead to a significant loss in functionality; hence, avoiding these inconsistencies is crucial. In addition, these inconsistencies are often difficult to detect, because (1) multiple model-view-controller groupings exist in the application, and (2) no exceptions are thrown or warnings provided in the event of an inconsistency.

To tackle this problem, we introduce an approach to automatically detect inconsistencies between identifiers in web applications developed using JavaScript MVC frameworks. Our design conducts static analysis to separate the three main components (model, view, and controller) in these applications; find the identifiers defined or used in these components; infer the types associated with these identifiers; and compare the collected identifiers and type information to determine any inconsistencies. We implement our approach in a tool called AUREBESH, which finds inconsistencies in AngularJS [2] applications, the most popular [6] JavaScript MVC framework used in practice.

Since MVC frameworks for JavaScript are fairly new, few papers have explored their characteristics. For the most part, prior work in this area does not include observations on the properties of existing MVC frameworks, but rather, proposes new MVC frameworks fitted towards a specific goal [7], [8], [9]. Other papers analyse existing JavaScript MVC frameworks, with particular focus on their maintainability [10], [11]. *To the best of our knowledge, our paper is the first to identify the consistency issues in JavaScript applications*

[1]*DOM* stands for *Document Object Model*, which is a data structure used to represent the hierarchy of HTML elements in the webpage and their properties.

using MVC frameworks,[2] *and the first to propose a design for automatically detecting inconsistencies in such applications.*

We list the following as our main contributions:

- We identify consistency issues pertinent to identifiers and types that are present in JavaScript MVC applications. These consistency issues point to potential problems within the application;
- We devise a general formal model for MVC applications. This model helps us reason about the way variables and functions are used and defined throughout the application, which, in turn, allows us to more clearly define what constitutes an inconsistency among them in the application;
- We introduce an automatic approach to detect identifier and type inconsistencies in MVC applications. This approach uses static analysis, and only requires the application's client-side source code;
- We implement our design in an open-source tool called AUREBESH, which works for AngularJS applications; and
- We perform a systematic fault injection experiment on AUREBESH to test its accuracy, and we subject AUREBESH to real-world applications to assess its ability to find real bugs. We find that AUREBESH is accurate (96.1% recall and 100% precision), and can find bugs in real MVC applications (15 bugs in 22 applications, five of which were acknowledged by the developers).

II. Running Example

The traditional application of MVC in web applications is to provide a clear separation between the application data and the HTML output that represents them on the server side. Recent JavaScript MVC frameworks represent the next logical step, i.e., applying the MVC model to the client-side to separate JavaScript (i.e., data and controls) from the DOM (i.e., the output).

Some popular MVC frameworks include AngularJS, BackboneJS, and Ember.js. Of these, AngularJS is the most widely used [6], with four times as many third-party modules and GitHub contributors, and over 20 times as many Chrome extension users, compared to the closest competitor, BackboneJS. Interest in AngularJS has also increased significantly since 2012, with around 50,000 questions in StackOverflow and 75,000 related YouTube videos. This is more than the corresponding items for the other two frameworks combined. For these reasons, we focus on AngularJS in this work.

We introduce the running example that we will be using throughout the paper. This example is inspired by real-world bugs encountered by developers of AngularJS applications [12], [13]. The application – which we will refer to as *MovieSearch* – initially takes the user to the "Search" page (Figure 1), where the user can input the name of a user, via the `input` element. Clicking on the "List User's Favourite Movies" button leads to the "Results" page (Figure 2), which displays the list of movies that corresponds to the user name

```
1 <input type="text" ng-model="userName" placeholder="Type
      Username" />
2 <button ng-click="searchUser()">
3   List User's Favourite Movies
4 </button>
```

Fig. 1. HTML code of the "Search" view (search.html)

```
1 <h3 ng-if="userData.display">
2   {{userData.intro}}
3 </h3>
4 <ul>
5   <li ng-repeat="movie in userData">
6     {{movie.name}}
7   </li>
8 </ul>
9 <div id="movieCount">
10   <ng-pluralize count="userData.count" when="movieForms"
        ></ng-pluralize>
11 </div>
12 <br />
13 <button ng-click="alertUserName()">
14   Which User?
15 </button>
```

Fig. 2. HTML code of the "Results" view (results.html)

that has been input, as well as the number of movies in the list. In addition, clicking on the "Which User?" button in the "Results" page would display the current user name in an alert box; for example, if the user name is "CountSolo", the alert would display the message, "The user is CountSolo".

The code for this application contains two views – one corresponding to the "Search" page (Figure 1) and the other corresponding to the "Results" page (Figure 2) – implemented in HTML. It also contains two models and two controllers implemented in JavaScript, shown in Figure 3.

An MVC application consists of *model variables*, *controller functions*, and *groupings*. Figure 5 shows how model variables and controller functions are defined and used, in relation to the models, views, and controllers. It also shows how these models, views, and controllers form groupings.

Model Variables. Model variables refer to the objects where the model data is stored, and are represented by identifiers defined within the scope of a particular model. These model variables are defined in models, and are used (either polled or updated) by associated views and controllers. For instance, the `Search` model in the running example defines one model variable, namely `userName` (Figure 3, line 4); further, the associated controller (`SearchCtrl`) and view (search.html) use this same variable in Figure 3, line 8, and Figure 1, line 1, respectively. Similarly, the `Results` model (Figure 3, lines 17-26) contains two model variables: `userData` and `movieForms`; these are used by the associated view (results.html) in various lines in Figure 2.

Controller Functions. Controller functions, as the name suggests, are functions defined in the controller. These controller functions are used in the view by attaching the function as an event handler to a view element. As an example, the `SearchCtrl` controller in Figure 3, lines 7-12 defines one controller function – `searchUser()` – which is subsequently used in the corresponding view (search.html) by setting it as

[2]For simplicity, we will refer to such applications as *MVC applications* or *JavaScript MVC applications*.

```
1 var searchApp = angular.module('searchApp', ['ngRoute'])↩
      ;
2 searchApp.controller('SearchCtrl', function($scope, ↩
      $location) {
3   //MODEL - Search
4   $scope.userName = "";
5
6   //CONTROLLER - SearchCtrl
7   $scope.searchUser = function() {
8     var id = getUserId($scope.userName);
9     if (id >= 0) {
10      $location.path('/results/' + id);
11    }
12  }
13 });
14
15 searchApp.controller('ResultsCtrl', function($scope, ↩
      $routeParams) {
16  //MODEL - Results
17  $scope.userData = {
18    movieList: getList($routeParams.userId),
19    intro: "Welcome User #" + $routeParams.userId,
20    display: true,
21    count: "two"
22  };
23  $scope.movieForms = {
24    one: '{} movie',
25    other: '{} movies'
26  };
27
28  //CONTROLLER - ResultsCtrl
29  $scope.alertUserName = function() {
30    alert("The user is " + $scope.userName);
31  };
32 });
```

Fig. 3. JavaScript code of the models and controllers

```
1 searchApp.config(function($routeProvider) {
2   $routeProvider
3     .when('/', {
4       controller: 'SearchCtrl',
5       templateUrl: 'search.html'
6     })
7     .when('/results/:userId', {
8       controller: 'ResultsCtrl',
9       templateUrl: 'results.html'
10    })
11    .otherwise({
12      redirectTo: '/'
13    });
14 });
```

Fig. 4. JavaScript code of the routes

the event handler of a button (Figure 1, line 2). Also, the `ResultsCtrl` controller in Figure 3, lines 29-31 defines the controller function `alertUserName()`, which is used in the corresponding view (results.html) in Figure 2, line 13.

Groupings. Due to the dynamic property of web applications, an MVC application can consist of multiple models, views, and controllers; hence, the programmer must specify which of these models, views, and controllers are associated with each other. Current MVC frameworks allow the programmer to specify these *(model, view, controller)* groupings by embedding the name of the model and controller in the view. These groupings can also be specified using *routers*, as in the case of the running example (Figure 4), which links the `Search` model and `SearchCtrl` controller with the search.html view (lines 3-6), and the `Results` model and `ResultsCtrl` controller with the results.html view (lines 7-10).

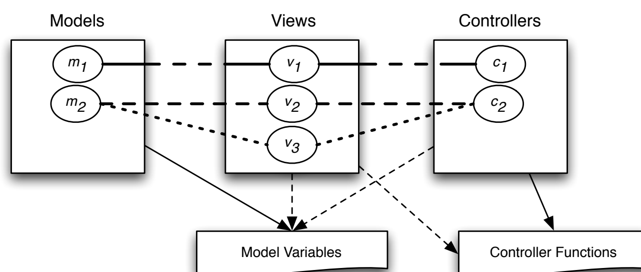

Fig. 5. Block diagram of the def-use and grouping model for MVC framework identifiers. Solid arrows indicate a "defines" relation, while dashed arrows indicate a "uses" relation. Models, views, and controllers connected with the same line types form a grouping.

III. Consistency Issues

We now describe two types of consistency issues observed in MVC applications, namely *identifier consistency* and *type consistency*. We focus on these issues in this paper.

Identifier Consistency. Model variables and controller functions are represented by identifiers in MVC applications. These identifiers are written both in the JavaScript code, when they are defined or used in the model or controller, and in the HTML code, when they are used in the view. To ensure correct operation, (1) model variable identifiers used in the controller or view must be *defined in the model*, and (2) controller function identifiers used in the view must be *defined in the controller*. While this seems straightforward to enforce at first sight, the following factors complicate the process of maintaining this consistency.

- An identifier is repeatedly used in both the HTML code and the JavaScript code. Even though DOM interactions are abstracted out by MVC frameworks, this repeated usage of identifiers across separate languages makes the application susceptible to identifier inconsistencies. Further, the common practice of implementing models, views, and controllers in separate files – sometimes maintained by separate programmers in collaborative projects – increases the chances of such inconsistencies.
- An application typically contains multiple models, views and controllers grouped together. Hence, the programmer must ensure the consistency not just of one *(model, view, controller)* grouping, but of several groupings. Also, these groupings must be set up correctly, e.g., via the routers, or else an inconsistency may occur.

For instance, the *MovieSearch* application contains two identifier inconsistencies. First, the `ResultsCtrl` controller uses the model variable identifier `userName` in Figure 3, line 30, but this identifier is not defined in the `Results` model (it is only defined in the `Search` model, which is not grouped with `ResultsCtrl`); this causes the alert box to display "The user is undefined" after clicking on the "Which User?" button. Second, since the `li` element in the results.html view loops over the `userData` model variable (Figure 2, lines 5-7) instead of `userData.movieList`, the reference to `movie.name` in

Figure 2, line 6 will be undefined with respect to the `Results` model; this causes blank bullet points to be displayed.

Type Consistency. In many cases, the programmer will also need to ensure that the value that is assigned to a model variable – or the value returned by a controller function – has a type consistent with that variable or function's use in the view. For example, in AngularJS, the `ng-if` attribute in the view must be assigned a Boolean value; a type inconsistency occurs if a model variable that contains a non-Boolean value or a controller function that returns a non-Boolean value is attached to the attribute. Ensuring this consistency is complicated by the fact that JavaScript is a loosely typed language.

MovieSearch contains one such type inconsistency. In Figure 2, line 10, the `userData.count` model variable is attached to the `count` attribute, which expects to be assigned a value of type Number; however, `userData.count` is assigned a String in the corresponding `Results` model (Figure 3, line 21). This leads to the disappearance of the message that shows the number of movies, inside the `div` element with ID `movieCount` in Figure 2.

IV. Formal Model of MVC Applications

We propose a more formal, abstract model for MVC-based web applications to clearly delineate all the consistency properties of such applications. This model also helps us describe our approach for automatically detecting inconsistencies.

Definition 1 (MVC Application)**.** *An* MVC application *is a tuple* $<\mathcal{M},\mathcal{V},\mathcal{C},\Omega,\Gamma,\omega_{\mathcal{M}},\omega_{\mathcal{V}},\omega_{\mathcal{C}},\gamma_{\mathcal{C}},\gamma_{\mathcal{V}},\phi>$ *where* $\mathcal{M}$ *is the set of models;* $\mathcal{V}$ *is the set of views;* $\mathcal{C}$ *is the set of controllers;* Ω *is the set of model variables; and* Γ *is the set of controller functions. Additionally, we define the following functions.*

- $\omega_{\mathcal{M}} : \mathcal{M} \rightarrow 2^{\Omega}$ *indicates what model variables are defined in a model;*
- $\omega_{\mathcal{V}} : \mathcal{V} \rightarrow 2^{\Omega}$ *indicates what model variables are used in a view;*
- $\omega_{\mathcal{C}} : \mathcal{C} \rightarrow 2^{\Omega}$ *indicates what model variables are used in a controller;*
- $\gamma_{\mathcal{C}} : \mathcal{C} \rightarrow 2^{\Gamma}$ *indicates what controller functions are defined in a controller;*
- $\gamma_{\mathcal{V}} : \mathcal{V} \rightarrow 2^{\Gamma}$ *indicates what controller functions are used in a view;*
- $\phi : \mathcal{M} \times \mathcal{V} \times \mathcal{C} \rightarrow \{true, false\}$ *indicates the model-view-controller groupings.*

Further, a model variable in Ω and a controller function in Γ are represented by a tuple $<id,ty>$, where *id* refers to the identifier, and *ty* refers to the type (for controller functions, this pertains to the *return* type). The function *I()* projects the *id* portion of these model variables and controller functions onto a set.

An MVC web application is *consistent* if and only if for every element $(m,v,c) \in \mathcal{M} \times \mathcal{V} \times \mathcal{C}$ such that $\phi(m,v,c) = true$, the following properties hold:

Property 1. The view and controller only use model variables that are defined in the model:

$$(\forall\mu)(\mu.id \in I(\omega_{\mathcal{C}}(c)) \cup I(\omega_{\mathcal{V}}(v)) \implies \mu.id \in I(\omega_{\mathcal{M}}(m)))$$

Property 2. The view only uses controller functions that are defined in the controller:

$$(\forall\kappa)(\kappa.id \in I(\gamma_{\mathcal{V}}(v)) \implies \kappa.id \in I(\gamma_{\mathcal{C}}(c)))$$

Property 3. The expected types of corresponding model variables in the view match the assigned types in the model or controller:

$$(\forall\mu,\rho)(\mu.id \in I(\omega_{\mathcal{V}}(v)) \wedge \rho.id \in I(\omega_{\mathcal{M}}(m)) \cup I(\omega_{\mathcal{C}}(c)) \wedge \mu.id = \rho.id \implies \mu.ty = \rho.ty)$$

Property 4. The expected and returned types of corresponding controller functions match in the view and controller.

$$(\forall\kappa,\tau)(\kappa.id \in I(\gamma_{\mathcal{V}}(v)) \wedge \tau.id \in I(\gamma_{\mathcal{C}}(c)) \wedge \kappa.id = \tau.id \implies \kappa.ty = \tau.ty)$$

V. Approach

To alleviate the consistency issues described, we propose a static analysis approach for automatically detecting identifier and type inconsistencies in MVC applications. We opt for a static instead of a dynamic approach for several reasons. First, static analysis is more lightweight than dynamic analysis, in that the application does not need to execute in order to detect the inconsistencies; this is especially useful during the development phase, where quick relay of information about the code, such as error messages, is preferred.

Second, dynamic analysis requires user input – i.e., a sequence of user events – and it is not always clear how to choose these inputs. A dynamic approach may be suitable for tools that target specific bugs – such as the JavaScript fault localization [14] and repair [15] tools we have previously developed – since the steps to reproduce the bug are known; in contrast, our detector is not targeting a specific bug known to exist in the program, but rather, *looking* for these bugs, without prior knowledge of how to reproduce them. This is the same reason an inconsistency detector is preferred over a mechanism that simply displays an error message when an inconsistency is encountered during execution.

There are also several challenges in designing the above detector, namely,

- **C1**: Model variables are often defined as nested objects (e.g., see Figure 3, lines 17-22), and the variables defined *inside* these objects, along with their types, also need to be recorded, thereby complicating the static analysis;
- **C2**: Sometimes, aliases are used in the HTML code to represent model variables defined in the JavaScript code (e.g., the `movie` variable in Figure 2, line 5 is an alias for `userData.movieList`, `userData.intro`, etc.). The design needs to be capable of handling these aliases;
- **C3**: Since MVC applications can contain multiple models, views, and controllers, the design needs to infer all the possible groupings to be checked; a simple comparison of *all* identifiers and types collected does not suffice.

Finally, our approach assumes that the code does not contain any instances of `eval`. This assumption is reasonable, as

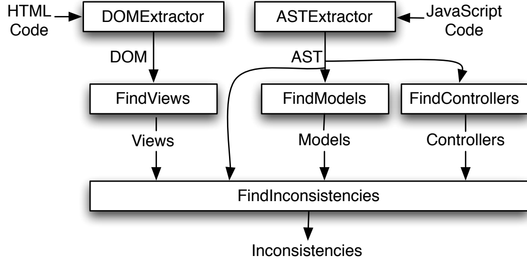

Fig. 6. Block diagram of our approach.

JavaScript MVC frameworks encourage programmers to write in a more declarative style; thus, features used in "vanilla" JavaScript such as `eval` are rarely seen in MVC applications.

A. Overview

The goal of our automatic inconsistency detector is to find all instances that violate Properties 1–4 in Section IV. The block diagram in Figure 6 shows an overview of our approach. As the figure depicts, the approach expects two inputs, namely the HTML (template) and the JavaScript code. The *DOMExtractor* converts the HTML template into its DOM representation, which is used to simplify analysis of the HTML elements and attributes. Similarly, the *ASTExtractor* converts the JavaScript code into its AST representation.

The modules *FindModels*, *FindViews*, and *FindControllers* statically analyze the DOM and the AST to populate the sets $\mathcal{M}$, $\mathcal{V}$, and $\mathcal{C}$, respectively. In our approach, we chose to represent a model $m \in \mathcal{M}$ as a tuple of the form $<name, ast>$, where *name* is a unique identifier for the model and *ast* is the subtree of the complete AST extracted earlier, containing only the nodes and edges pertinent to the model; for example, the value of *ast* for the `Results` model in Figure 3 would be the AST representing lines 17-26. Similarly, a view $v \in \mathcal{V}$ and a controller $c \in \mathcal{C}$ are represented by $<name, dom>$ and $<name, ast>$, respectively. Section V-B describes in more detail how the above sets are populated.

Once $\mathcal{M}$, $\mathcal{V}$, and $\mathcal{C}$ are all populated, these sets, along with the complete DOM and AST, are input into the *FindInconsistencies* module (see Algorithm 1). The output of this algorithm is a list of inconsistencies Q. It starts by initializing the sets ϕ, as well as the "identifier inclusion functions" $\omega_{\mathcal{M}}$, $\omega_{\mathcal{V}}$, $\omega_{\mathcal{C}}$, $\gamma_{\mathcal{C}}$, and $\gamma_{\mathcal{V}}$ with the contents of $\mathcal{M}$, $\mathcal{V}$, and $\mathcal{C}$ (lines 1-7); here, all the models, views, and controllers initially map to the empty set, since the model variables and controller functions are still not known. These mappings are updated as identifiers are discovered by the *findIdentifiers()* function, described in Section V-C. Likewise, the mappings in ϕ are updated, in line 9, by the *findMVCGroupings()* function (Section V-D). Lines 10-27 are responsible for detecting the identifier and type mismatches, and are described in detail in Section V-E.

B. Finding the Models, Views and Controllers

The *FindModels*, *FindViews*, and *FindControllers* modules in Figure 6 populate $\mathcal{M}$, $\mathcal{V}$, and $\mathcal{C}$, respectively, by locating the corresponding structures or blocks in the HTML and JavaScript code. For example, in AngularJS, models and controllers are added as the body of the function passed to the `.controller()` method as a parameter (see Figure 3). Hence, in this case, locating the models and controllers involves finding the subtrees in the AST that are rooted at a `CallExpression` for the method `.controller()`, and parsing the body of the function parameter. Similarly, views are normally saved as separate HTML files, so in most cases, finding them is tantamount to identifying these separate files.

Algorithm 1: FindInconsistencies

Input: $\mathcal{M}$: The set of models
Input: $\mathcal{V}$: The set of views
Input: $\mathcal{C}$: The set of controllers
Input: *DOM*: The complete DOM
Input: *AST*: The complete AST
Output: Q: List of inconsistencies

1. $Q \leftarrow \emptyset$;
2. $\phi \leftarrow \{((m,v,c), false) \mid m \in \mathcal{M} \wedge v \in \mathcal{V} \wedge c \in \mathcal{C}\}$;
3. $\omega_{\mathcal{M}} \leftarrow \{(m,\emptyset) \mid m \in \mathcal{M}\}$;
4. $\omega_{\mathcal{V}} \leftarrow \{(v,\emptyset) \mid v \in \mathcal{V}\}$;
5. $\omega_{\mathcal{C}} \leftarrow \{(c,\emptyset) \mid c \in \mathcal{C}\}$;
6. $\gamma_{\mathcal{C}} \leftarrow \{(c,\emptyset) \mid c \in \mathcal{C}\}$;
7. $\gamma_{\mathcal{V}} \leftarrow \{(v,\emptyset) \mid v \in \mathcal{V}\}$;
8. $findIdentifiers(\mathcal{M}, \mathcal{V}, \mathcal{C}, \omega_{\mathcal{M}}, \omega_{\mathcal{V}}, \omega_{\mathcal{C}}, \gamma_{\mathcal{C}}, \gamma_{\mathcal{V}})$;
9. $\phi \leftarrow findMVCGroupings(\mathcal{M}, \mathcal{V}, \mathcal{C}, DOM, AST)$;
10. **foreach** $(m,v,c) \in \{(m,v,c) \mid \phi(m,v,c) = true\}$ **do**
11.   **foreach** $mv \in \omega_{\mathcal{V}}(v) \cup \omega_{\mathcal{C}}(c)$ **do**
12.   **if** $mv.id \notin I(\omega_{\mathcal{M}}(m))$ **then**
13. $Q \leftarrow Q \cup \{idMismatch(mv)\}$;
14.   **end**
15.   **else if** $!matchingType(mv, \omega_{\mathcal{M}}(m))$ **then**
16. $Q \leftarrow Q \cup \{typeMismatch(mv)\}$;
17.   **end**
18.   **end**
19.   **foreach** $cf \in \gamma_{\mathcal{V}}(v)$ **do**
20.   **if** $cf.id \notin I(\gamma_{\mathcal{C}}(c))$ **then**
21. $Q \leftarrow Q \cup \{idMismatch(cf)\}$;
22.   **end**
23.   **else if** $!matchingType(cf, \gamma_{\mathcal{C}}(c))$ **then**
24. $Q \leftarrow Q \cup \{typeMismatch(cf)\}$;
25.   **end**
26.   **end**
27. **end**

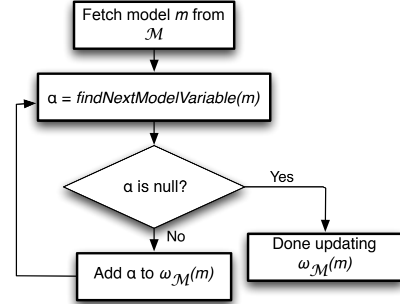

Fig. 7. Portion of *findIdentifiers* that updates $\omega_{\mathcal{M}}$ for every model. The other "identifier inclusion functions" are updated in a similar way.

C. Inferring Identifiers

The goal of the *findIdentifiers* module, which is invoked in Algorithm 1, line 8, is to find the model variables and controller functions that are defined or used in every model, view, and controller that were found earlier (see Section V-B), thereby updating the mappings in the "identifier inclusion functions". Figure 7 illustrates how *findIdentifiers* looks for model variables defined in every model. A similar algorithm is used to find the model variables and controller functions used or defined in other entities.

The functions *findNextModelVariable* and *findNextControllerFunction* analyze the DOM and the AST according to the syntactic styles imposed by the MVC framework being used. In AngularJS, model variables are defined as a property of the `$scope` variable in an assignment expression (see Figure 3, lines 4, 17, and 23); controller functions are defined similarly, albeit the right side of the expression is a Function object (e.g., Figure 3, lines 7 and 29). Finding identifiers used in views, however, is trickier – although identifiers appear as attribute values of DOM elements in many cases, they also typically appear in double curly brace notation as part of a text element (e.g., Figure 2, lines 2 and 6); hence, text elements in the view's DOM are also parsed since these may contain references to identifiers.

Type Inference. To find the type assigned to a model variable in the model or controller, our approach looks at the right-hand side of the assignment expression and infers the assigned type based on the AST node (e.g., if the right-hand side is a `StringLiteral` node, the inferred type is String). If the expression is too complicated and the assigned type cannot be inferred, the type is recorded as *unknown* for that identifier, so our type inference algorithm is conservative. The simplification we made for type inference requires the assigned expression to be a literal.[3] Although this may seem to be a significant limitation, note that simple assignments are commonplace in MVC applications, perhaps because MVC frameworks are designed such that applications can be programmed in a "declarative" way [16]; hence, we believe our simplification is justified (we further validate this claim empirically in Section VII). Note that type inference works similarly for controller functions, except that the *return value expressions* are parsed instead of assignments.

To infer the expected type of a model variable or controller function used in a view, our approach examines the attribute to which the identifier is assigned and determines if this attribute expects values belonging to a specific type. For instance, the `count` attribute in AngularJS expects a Number, so this is recorded as the expected type for `userData.count` in Figure 2, line 10. If the identifier has no expected types, its type is simply recorded as $\perp$, which matches all types.

There are two special cases that our algorithm must handle, namely, nested objects and aliases.

Nested Objects. To address challenge **C1** (see beginning of Section V), we also model nested objects such as the `userData` and `movieForms` variables in Figure 3. Our approach represents nested objects as a tree. Each node in the tree represents an identifier, with an assigned type. The trees for the `userData` and `movieForms` variables are joined together in one root as they both belong to the same model. This is shown in Figure 8.

Analogously, if a model variable in the view uses the dot notation, then it is represented as a sequence of identifiers. For

[3]Our design performs some "smart parsing", e.g., it can detect concatenations of strings, but we omit their description due to space limitations.

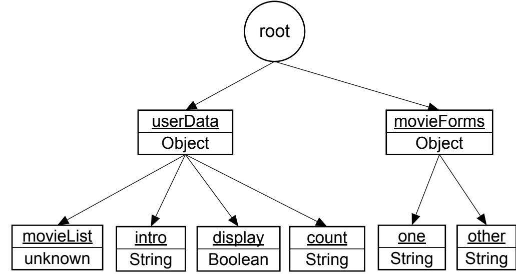

Fig. 8. Tree representing the model variables in the `Results` model of *MovieSearch*, including all the nested objects. The identifiers are shown at the top of each node, while the types are shown at the bottom.

example, `userData.count` in Figure 2, line 10 is represented as `root` $\rightarrow$ `userData` $\rightarrow$ `count`, with expected type Number.

Aliases. An example of the use of aliases (challenge **C2**) is the `ng-repeat` directive in AngularJS, which replicates the associated HTML element in the DOM for each element of some specified collection. This directive is assigned a string value of the form "<alias> in <collection>", where <collection> is an array or an object, and <alias> is an identifier that represents each member of the array (or each property of the object) in each replication of the HTML element in the DOM.

Figure 2, line 5 shows an example, in which the collection is the `userData` object and the alias is `movie`. Therefore, the alias `movie` refers to every property of `userData`, namely `userData.movieList`, `userData.intro`, `userData.display`, and `userData.count`. Subsequently, the reference to `movie.name` in Figure 2, line 6, translates to each of these four identifiers, followed by "`.name`". These four sequences are therefore included as model variables in the results.html view – i.e., they are all added in the list that maps to the results.html view when updating $\omega_{\mathcal{V}}$.

D. Discovering MVC Groupings

At this point, the model variables and controller functions have been discovered, and have been mapped to their associated models, views, and controllers. Our approach must now find all model-view-controller combinations that can potentially appear in the application, to address challenge **C3**. More formally, our approach must find all $(m, v, c) \in \mathcal{M}x\mathcal{V}x\mathcal{C}$ such that $\phi(m, v, c) = true$, thereby updating ϕ in the process.

This procedure is carried out by the *findMVCGroupings()* function; as seen in Algorithm 1, line 9, this function takes $\mathcal{M}$, $\mathcal{V}$, and $\mathcal{C}$ as inputs, along with the complete AST and DOM. The reason the full AST and DOM are needed is that *findMVCGroupings()* will look for information in the DOM that explicitly maps a specific model or controller to a view via an HTML attribute, as well as routing information in the AST that does the same. This information, coupled with the *name* part of each model, view, and controller, allows our approach to determine all the valid groupings.

Take, for example, the router for *MovieSearch* in Figure 4. The first route (lines 3-7) groups the `Results` model, the `ResultsCtrl` controller, and the `results.html` view. Thus,

ϕ is updated so that the model, view, and controller objects with these respective identifiers together map to *true*. In other words, if *m.name* = `Results`, *v.name* = `results.html` and *c.name* = `ResultsCtrl`, then the design sets $\phi(m,v,c) = true$. This process is repeated for all other groupings discovered.

E. Detecting Inconsistencies

The final step in our approach is to compare the model variables and controller functions within the same grouping and detect any potential inconsistencies The pseudocode for this procedure is shown in Algorithm 1, lines 10-27.

The algorithm begins by looking for inconsistencies related to model variables (lines 11-18). Line 11 loops through every model variable used in either the view or the controller. For all such model variables *mv*, the *id* is checked to see if it also exists among the model variables defined in the corresponding model (line 12). If not, this means that Property 1 is violated and there is an identifier inconsistency, so this inconsistency is included in Q. However, if the *id* does exist in the model, the matching model variable in the model is compared with *mv* to see if they have the same type. If the types do not match, then Property 3 is violated and there is a type inconsistency; this inconsistency is then included in Q. The algorithm for finding inconsistencies in controller functions (lines 19-26) is similar. Note that model variables with unknown types are assumed to match all types. The remaining question, then, is, how are the identifier and type comparisons made? For controller functions, the answer is straightforward – identifiers are compared based on a string comparison, and types are compared based on the assigned and returned types that were previously inferred in Section V-C.

Model variables are, however, more challenging because of the possibility of nested objects. In this case, the sequence representation of *mv* is used to traverse the tree representing the model variables defined in the corresponding model. Take, for instance, the sequence `root` $\rightarrow$ `userName`, which is used in the `ResultsCtrl` controller, as per Figure 3, line 30. If this sequence is used to traverse the tree representing the model variables defined in `Results` (see Figure 8) starting from the root, our design will discover that the given sequence does not exist in the tree, and therefore, there is an identifier inconsistency. In addition, the sequence `root` $\rightarrow$ `userData` $\rightarrow$ `count` is used in the results.html view, as per Figure 2, line 10, and has an expected type of Number since it is assigned to the `count` attribute in `ng-pluralize`. If this sequence is used to traverse the same tree, the traversal will be successful, since the sequence exists in the tree. However, note that the expected type of the terminating node in the traversal (*count*) is String, which does not match the expected type of Number. Thus, a type inconsistency is recorded for this sequence. Finally, if `root` $\rightarrow$ `userData` $\rightarrow$ `intro` $\rightarrow$ `name` – which is one of the sequences the `movie.name` alias translated to as described in Section V-C – is used to traverse the tree, the traversal will fail, since the sequence does not exist in the tree. As a result, another identifier inconsistency will be recorded. In summary, our design is able to detect all the three inconsistencies in the *MovieSearch* running example.

TABLE I
REAL BUGS FOUND. THE "FAULT TYPE" COLUMN REFERS TO THE FAULT TYPE NUMBER, AS PER TABLE II.

Application	Fault Type	Error Message	Severity
Cafe Townsend	2	Undefined model variable `employee.id`	3
	2	Undefined model variable `employee.id`	3
Cryptography	3	Undefined model variable `lastWordCount`	1
Dematerializer	3	Undefined model variable `editing`	3
eTuneBook	5	Undefined controller function `doneTuneSetEditing`	3
	7	Inconsistent type for `currentFilter`	3
Flat Todo	4	Undefined model variable `showTaskPanel`	5
	4	Undefined model variable `showStatusFilter`	5
GQB	7	Inconsistent type for `download.aggregated`	3
Hackynote	3	Undefined model variable `theme.current.css`	4
	3	Undefined model variable `transition.current.css`	4
Reddit Reader	3	Undefined model variable `post`	2
Story Navigator	1	Undefined model variable `ui.columns.status`	3
beLocal	3	Undefined model variable `likeDisabled`	3
Linksupp	3	Undefined model variable `startEating`	2

VI. IMPLEMENTATION

We implemented our approach in a tool called AUREBESH. It is built on top of the Ace Editor, which is the editor used for the Cloud9 IDE [17]. We have embedded Ace Editor as part of a web application that can be accessed in our website [18]. AUREBESH is implemented entirely using JavaScript, and currently supports MVC applications written with AngularJS.

To invoke the detector, we added a "Find Inconsistencies" button to the IDE, which the user must click. For every inconsistency found by the detector, an error message is highlighted on the line of code containing the inconsistency in the IDE. The user can then click on these error messages to get more details about the inconsistencies.

VII. EVALUATION

To assess the efficacy and real-world relevance of our approach, we address the following research questions:

RQ1 (Real Bugs): Can AUREBESH help developers to find bugs in real-world MVC applications?

RQ2 (Accuracy): What is the accuracy of AUREBESH in detecting identifier and type inconsistencies in MVC applications?

RQ3 (Performance): How quickly can AUREBESH perform the inconsistency detection analysis?

A. Subject Systems

In total, we consider 20 open-source AngularJS applications for our experiments, listed in Table III. These applications were chosen from a list of MVC applications from AngularJS'

GitHub page [19]; in particular, only the applications whose source code is available and unobfuscated are considered, since AUREBESH, in its current state, is incapable of working with obfuscated code. This is not a fundamental limitation though as AUREBESH is meant for developers to use before obfuscating their code. As shown in Table III, the applications cover a variety of sizes and application types.

B. Methodology

Real Bugs. To answer RQ1, we run AUREBESH on the 20 subject systems. For this experiment, we also ran our tool on two additional AngularJS applications developed by students for a software engineering course at the University of Victoria [20], namely beLocal and Linksupp. We analyze every error message reported by AUREBESH for these applications to see if it corresponds to a real bug. We report any true positives (i.e., error messages that correspond to real bugs) and false positives (i.e., spurious error messages) that we find.

Accuracy. To measure the accuracy (RQ2), we conduct a fault injection study on the subject systems. An injection is performed on an application by introducing a mutation to a line of code from one of its source files (either the HTML or JavaScript code), running AUREBESH on this mutated version of the application, and then recording if AUREBESH detects the inconsistency introduced by the mutation. If the inconsistency is detected, the result of the injection is marked as "successful"; otherwise, the injection is marked as "failed".

In this experiment, we consider ten types of mutations, as seen in Table II. The "expected behaviour" for a mutation describes the correct error message that AUREBESH is expected to display when running AUREBESH on an application with this mutation applied. Each of these mutation types corresponds to a *violation* of one of the four properties listed in Section IV; hence, the results for a mutation type give an indication of how well AUREBESH detects violations of the corresponding property.

For each application, we perform 20 injections per mutation type, which amounts to a total of 200 injections per application. However, note that a mutation type may not be applicable for certain applications (e.g., not all controllers use model variables, in which case mutation type #2 will not be applicable); this explains why several applications have fewer than 200 injections (see Table III). The specific location mutated in the code is chosen uniformly at random from among the lines of code applicable to the current mutation type. For each injection, we record the number of successful detections, the number of failed detections, and the number of spurious error messages introduced by the mutation; this allows us to report both the recall (the number of successful detections over the total number of injections) and precision (the number of successful detections over the total number of error messages displayed) of AUREBESH.

Performance. We measure performance by running AUREBESH on each subject system, recording the analysis completion time and averaging over multiple trials.

TABLE II
TYPES OF FAULTS INJECTED. MV REFERS TO "MODEL VARIABLE", AND CF REFERS TO "CONTROLLER FUNCTION".

Type #	Description	Expected Behaviour	Property Tested
1	Modify the name of a MV used in *line N* of a view	Detect undefined MV in *line N*	1
2	Modify the name of a MV used in *line N* of a controller	Detect undefined MV in *line N*	1
3	For a particular MV used in *line N* of a view, delete the definition of that MV in a corresponding model	Detect undefined MV in *line N*	1
4	For a particular MV used in *line N* of a controller, delete the definition of that MV in a corresponding model	Detect undefined MV in *line N*	1
5	Modify the name of a CF used in *line N* of a view	Detect undefined CF in *line N*	2
6	For a particular CF used in *line N* of a view, delete the definition of that CF in a corresponding controller	Detect undefined CF in *line N*	2
7	For a particular MV used in the view that expects a certain type *T1*, modify the definition of that MV in *line N* of a corresponding model so that the type is changed to *T2*	Detect type mismatch in *line N* (*T1* expected but type is *T2*)	3
8	For a particular MV used in the view that expects a certain type *T1* and defined in *line N* of a corresponding model, modify the expected type to *T2* by mutating the `ng` attribute name	Detect type mismatch in *line N* (*T2* expected but type is *T1*)	3
9	For a particular CF used in the view that expects a certain type *T1*, modify the return value of that CF in *line N* of the controller to a value of type *T2*	Detect type mismatch in *line N* (*T1* expected but type is *T2*)	4
10	For a particular CF used in the view that expects a certain type *T1* and returns a value in *line N* of a corresponding controller, modify the expected type to *T2* by mutating the `ng` attribute name	Detect type mismatch in *line N* (*T2* expected but type is *T1*)	4

C. Results

Real Bugs. After running AUREBESH on the original, unaltered versions of the subject systems, AUREBESH displayed a total of 15 error messages in 11 applications. We reported these error messages to the developers, and five of them (the error messages from Story Navigator, beLocal, Linksupp, and two from Hackynote) were acknowledged as real issues and fixed. The other applications, unfortunately, are no longer maintained by the developers, so our bug reports for those applications remain unacknowledged. Nonetheless, we analyzed the 15 error messages and found that they are all true positives i.e., they all correspond to real-world bugs.

Of the 15 bugs, we found 13 identifier inconsistencies and 2 type inconsistencies; as shown in Table I, each of the bugs our tool found maps to one of the fault types in Table II. Note that the two error messages in Cafe Townsend, while identical, correspond to two different bugs. With regards to *why* these faults were committed, we identified the following patterns:

- **Identifier defined elsewhere** (7 cases): There are several cases where assignments representing the model variable definitions are placed not in the model itself, but inside controller functions. This applies, for example, to the model variable `lastWordCount` in Cryptography;
- **Incorrect identifier** (5 cases): In some cases, the inconsistencies arise because the programmer has typed incorrect identifiers. For instance, in Hackynote, the identifier

TABLE III
FAULT INJECTION RESULTS. THE SIZE PERTAINS TO THE COMBINED LINES OF HTML AND JAVASCRIPT CODE, NOT INCLUDING LIBRARIES.

Application	Application Category	Size (LOC)	Successful Detections	Failed Detections	Total Injections	Recall (%)	Precision (%)
Angular Tunes	Music Player	185	40	0	40	100.00	100.00
Balance Projector	Finance Tracker	511	140	20	160	87.50	100.00
Cafe Townsend	Employee Tracker	452	160	0	160	100.00	100.00
CodeLab	RSS Reader	602	79	1	80	98.75	100.00
Cryptography	Encoder	523	120	0	120	100.00	100.00
Dematerializer	Blogging	379	186	14	200	93.00	100.00
Dustr	Template Compiler	493	80	0	80	100.00	100.00
eTuneBook	Music Manager	5042	177	23	200	88.50	100.00
Flat Todo	Todo Organizer	255	107	13	120	89.17	100.00
GitHub Contributors	Search	459	142	18	160	88.75	100.00
GQB	Graph Traversal	1170	194	6	200	97.00	100.00
Hackynote	Slide Maker	236	120	0	120	100.00	100.00
Kodigon	Encoder	948	120	0	120	100.00	100.00
Memory Game	Puzzle	181	40	0	40	100.00	100.00
Pubnub	Chat	134	120	0	120	100.00	100.00
Reddit Reader	Reader	255	120	0	120	100.00	100.00
Shortkeys	Shortcut Maker	407	120	0	120	100.00	100.00
Sliding Puzzle	Puzzle	608	40	0	40	100.00	100.00
Story Navigator	Test Case Tracker	415	117	3	120	97.50	100.00
TwitterSearch	Search	357	199	1	200	99.50	100.00
OVERALL			2421	99	2520	96.07	100.00

given for a property in the nested object `theme.current` is `src`, but the identifier expected by the view is `css`;

- **Boolean assigned a string** (2 cases): The two type inconsistencies involved the programmer assigning a string to a model variable that expects a boolean value. For instance, in GQB, the `download.aggregated` variable was erroneously assigned the string value "true" instead of the boolean value `true`;
- **Identifier name not updated** (1 case): This occurs in eTuneBook. Upon inspection, it turned out that the undefined controller function `doneTuneSetEditing` in eTuneBook was defined in previous versions of the application, but was replaced with another function with a different name; the reference to the old name remained in the view. This is an example of a regression bug.

Table I also shows the severity of the bugs, based on our qualitative assessment of these bugs; here, we use Bugzilla's ranking scheme, where 1 represents the lowest and 5 represents the highest severity. Although some of the bugs are cosmetic (e.g., the bug in Cryptography simply causes one of the labels to display as "One of possible –word permutations...", with the number next to "–word" missing), many of the bugs have considerable impact on the application. For example, the first bug in Flat Todo renders the "plus" button – which adds todos in the list – useless. A similar effect takes place in eTuneBook, where the missing controller function makes one of the buttons inoperable, thereby preventing the user from exiting edit mode. Also, the two bugs in Hackynote prevented the user from removing the theme and transition present in the slides.

Lastly, AUREBESH displayed only one false positive, in the Linksupp application. The reason is that the application uses the `$rootScope` variable to define model variables to be within the scope of *all* models. Our tool assumes that every model variable used in a view is defined only via the `$scope` variable, leading to the false positive. Nonetheless, the low number of false positives indicates that the error messages displayed by our tool are trustworthy, minimizing the effort required to filter out any spurious messages.

Accuracy. Table III shows the per-application results for the fault injection experiment. As the table shows, AUREBESH is very accurate, yielding an overall recall of 96.1%, and attains a perfect recall for eleven of the twenty applications. In addition, AUREBESH did not output any spurious messages during any of the injections; hence, AUREBESH *was able to attain an overall precision of 100% in this experiment*.

To understand what is causing the failed detections, we divide the results in terms of the properties (Section IV) being violated by the mutation types. As seen in Table IV, Properties 1, 3 and 4 have imperfect recalls. We analyzed the 27 failed detections for Property 1, which represents the consistency of model variable identifiers, and found that they all result from the usage of "filters" in conjunction with model variables in views. These filters are used in AngularJS to customize the appearance of model variables' values when displayed by the view; AUREBESH currently does not recognize these filters and ignores them when parsing, leading to the failed detection. Note that this limitation is implementation specific, and can be overcome by extending the parser.

We also analyzed the 72 failed detections for Properties 3 and 4, both of which represent the consistency of types. We found that these are caused by our assumption that the values assigned by model variables or returned by controller functions are either literals or simple expressions, and thus have types that are easy to infer. More specifically, in these 72 cases, the values assigned or returned are either complex expressions or retrieved from an external database. This prevented AUREBESH from inferring the types; since AUREBESH is conservative, it does not report the type inconsistency. Overall, these cases constitute 14% of the cases where type inference was needed. Note that in the remaining 448 cases (86% of the cases), the values were literals or simple expressions, which indicates that our assumption is valid in the majority of cases.

TABLE IV
FAULT INJECTION RESULTS PER PROPERTY.

Property	Successful Detections	Failed Detections	Total Injections	Recall (%)
1	1293	27	1320	98.0
2	560	0	560	100.0
3	268	52	320	83.8
4	180	20	200	90.0

Performance. For each subject system, AUREBESH was able to perform the analysis in an average time of 121 milliseconds, with a worst-case of 849 milliseconds for the largest application, eTuneBook. This indicates that performance is not an issue with our tool.

VIII. DISCUSSION

1) Limitations: The implementation of our approach for AngularJS has a few limitations. First, as explained in Section VII-C, AUREBESH currently disregards the presence of filters in views in AngularJS. Also, as mentioned in Section VII-C, our tool currently disregards the use of the `$rootScope` variable, which can lead to false positives.

With respect to the approach itself, a limitation is in our type inference algorithm, which assumes simple assignments and return values. Our results suggest that this assumption is reasonable; however, we also found that a considerable number (around 14%) of pertinent assignments and return values involve complex expressions or external database accesses, so a more advanced type inference algorithm is needed. Lastly, AUREBESH does not consider inheritance in MVC applications, where models are made descendants of other models to allow model variables to be inherited. Again, we have not encountered this in practice, but it can occur.

Another limitation of AUREBESH is that it works only on applications written using AngularJS. While AngularJS is the most popular client-side MVC framework, our problem formulation (Section III), formal model (Section IV) and algorithm (Section V) are all fairly generic and can be applied to other MVC frameworks with minimal modifications.

2) Threats to Validity: One internal validity threat regards the mutation types used in our fault injection experiment, and how representative they are of both our inconsistency model and real-world bugs. To address this issue, we selected the mutation types such that they all map to the consistency properties presented in Section IV. In addition, each of the 15 real-world bugs that we found in one of our experiments maps to a mutation type, as described in Section VII-C, giving an indication of the mutation types' representativeness.

As with any experiment that considers a limited number of subject systems, the generalizability of our results may be called into question, which is an external threat to validity. Unfortunately, since AngularJS is a fairly new framework, applications using this framework are quite scarce. Fortunately, the AngularJS GitHub page provides a list of web applications using that framework; to mitigate the external threat, we chose applications of different types and sizes from this list.

Finally, the source code of all the subject systems we considered in our experiments are all available online; further, we kept our own records of the source code of these systems that AUREBESH analyzed. Our tool AUREBESH, is also publicly available. Hence, our experiments are reproducible.

IX. RELATED WORK

MVC has been applied to various domains, and one of its earliest uses can be traced back to Xerox PARC's Smalltalk [21]. The pattern has also been applied to the server-side of web applications [22], where the model and controller are implemented on the server and the view is represented by the HTML output on the client. Since the application of MVC to client-side web application programming is a fairly recent development, there are only a few papers addressing this topic. Much of the research in this area has focused on the application of the MVC model to JavaScript development, tailored towards specific application types [7], [8], [9]. Studies on JavaScript MVC frameworks' properties have been limited to an analysis of their maintainability [10], [11]. Unlike our present work, these studies do not consider the presence of consistency issues in JavaScript MVC applications, nor do they propose an approach for analyzing MVC application code.

Several papers have analyzed the characteristics of common JavaScript frameworks, such as jQuery [23], [24], [25], [26]. Richards et al. [27] and Ratanaworabhan et al. [28] analyze the effect that different frameworks have on the dynamic behaviour of JavaScript in web applications. Feldthaus and Møller [29] look at TypeScript interfaces, and propose a tool that checks for the correctness of these interfaces. In prior work, we also briefly explored the relationship between JavaScript frameworks and JavaScript faults that occur in production websites [30]. Our current work differs from these studies in that they consider non-MVC frameworks, which have different usage patterns compared to MVC frameworks.

Finally, considerable work has been done on the application of MVC on the server-side [31], [32], [33], [34], where frameworks such as Spring MVC and JSF are used. Wojciechowski et al. [35] compared different MVC-based design patterns on the server-side, and analyzed the frameworks' characteristics, such as their susceptibility to file upload issues. In contrast, our work is concerned with the client-side of web applications.

X. CONCLUSION AND FUTURE WORK

In this paper, we presented an automated technique and tool AUREBESH, which statically analyzes applications written using AngularJS, a popular JAVASCRIPT MVC framework, to detect inconsistencies in the code. Our evaluation of AUREBESH indicates that it is accurate, with an overall recall of 96.1% and a precision of 100%. We also find that it is useful in finding bugs in MVC applications – in total, we found 15 real-world bugs in 22 AngularJS web applications.

ACKNOWLEDGMENT

This research was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), and a research gift from Intel Corporation. We thank Arie van Deursen for his invaluable feedback on our work.

REFERENCES

[1] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah, "An empirical study of client-side JavaScript bugs," in *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE Computer Society, 2013, pp. 55–64.

[2] "AngularJS," http://www.angularjs.org.

[3] "BackboneJS," http://www.backbonejs.org.

[4] "EmberJS," http://www.emberjs.com.

[5] E. Koshelko, "Why you should not use AngularJS," 2015, https://medium.com/@mnemon1ck/why-you-should-not-use-angularjs-1df5ddf6fc99.

[6] U. Shaked, "AngularJS vs. BackboneJS vs. EmberJS," 2014, http://www.airpair.com/js/javascript-framework-comparison.

[7] B. Taraghi and M. Ebner, "A simple MVC framework for widget development," in *Proceedings of the International Workshop on Mashup Personal Learning Environments (MUPPLE)*. CEUR-WS, 2010, pp. 38–45.

[8] Y. Hongping, S. Jiangping, and Z. Xiaorui, "The update version development of "wiki message linking" system-integrated Ajax with MVC model," in *Proceedings of the International Forum on Computer Science-Technology and Applications (IFCSTA)*. IEEE Computer Society, 2009, pp. 209–212.

[9] J. Fujima, "Building a meme media platform with a JavaScript MVC framework and HTML5," *Webble Technology*, pp. 79–89, 2013.

[10] K. Kambona, E. G. Boix, and W. De Meuter, "An evaluation of reactive programming and promises for structuring collaborative web applications," in *Proceedings of the Workshop on Dynamic Languages and Applications (DYLA)*. ACM, 2013, pp. 15–23.

[11] V. Balasubramanee, C. Wimalasena, R. Singh, and M. Pierce, "Twitter bootstrap and AngularJS: Frontend frameworks to expedite science gateway development," in *Proceedings of the International Conference on Cluster Computing (CLUSTER)*. IEEE Computer Society, 2013, p. 1.

[12] "Scope variable not accessible (undefined) - AngularJS," 2014, http://www.jjask.com/554996/scope-variable-not-accessible-undefined-angularjs.

[13] C. Robinson, "AngularJS: If you don't have a dot, you're doing it wrong!" 2013, http://zcourts.com/2013/05/31/angularjs-if-you-dont-have-a-dot-youre-doing-it-wrong/.

[14] F. Ocariza, K. Pattabiraman, and A. Mesbah, "AutoFLox: an automatic fault localizer for client-side JavaScript," in *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, 2012, pp. 31–40.

[15] F. Ocariza, K. Pattabiraman, and A. Mesbah, "Vejovis: suggesting fixes for JavaScript faults," in *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 837–847.

[16] "What is AngularJS?" https://docs.angularjs.org/guide/introduction.

[17] "Ace," http://ace.c9.io/.

[18] "Aurebesh," http://ece.ubc.ca/~frolino/projects/aurebesh/.

[19] "Built with AngularJS," https://github.com/angular/builtwith.angularjs.org/blob/master/projects/projects.json.

[20] "CSC485B: Startup Programming," https://github.com/alexeyza/startup-programming.

[21] G. E. Krasner and S. T. Pope, "A cookbook for using the model-view controller user interface paradigm in Smalltalk-80," *Journal of Object Oriented Program (JOOP)*, vol. 1, no. 3, pp. 26–49, 1988.

[22] A. Leff and J. T. Rayfield, "Web-application development using the model/view/controller design pattern," in *Proceedings of the International Enterprise Distributed Object Computing Conference (EDOC)*. IEEE Computer Society, 2001, pp. 118–127.

[23] A. Gizas, S. Christodoulou, and T. Papatheodorou, "Comparative evaluation of javascript frameworks," in *Proceedings of the International Conference Companion on World Wide Web (WWW Companion)*. ACM, 2012, pp. 513–514.

[24] Y. Liao, Z. Zhang, and Y. Yang, "Web applications based on AJAX technology and its framework," in *Proceedings of the International Conference on Communications and Information Processing (ICCIP)*. Springer, 2012, pp. 320–326.

[25] D. Graziotin and P. Abrahamsson, "Making sense out of a jungle of JavaScript frameworks," *Product-Focused Software Process Improvement (PROFES)*, pp. 334–337, 2013.

[26] V. Y. Rosales-Morales, G. Alor-Hernández, and U. Juárez-Martínez, "An overview of multimedia support into JavaScript-based frameworks for developing rias," in *Proceedings of the International Conference on Electrical Communications and Computers (CONIELECOMP)*. IEEE Computer Society, 2011, pp. 66–70.

[27] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of JavaScript programs," in *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2010, pp. 1–12.

[28] P. Ratanaworabhan, B. Livshits, D. Simmons, and B. Zorn, "JSMeter: Measuring JavaScript behavior in the wild," in *Proceedings of the USENIX Conference on Web Application Development (WebApps)*. ACM, 2010, pp. 1–12.

[29] A. Feldthaus and A. Møller, "Checking correctness of TypeScript interfaces for JavaScript libraries," in *Proceedings of the International Conference on Object Oriented Programming, Systems, Language and Applications (OOPSLA)*. ACM, 2014.

[30] F. Ocariza, K. Pattabiraman, and B. Zorn, "JavaScript errors in the wild: an empirical study," in *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Computer Society, 2011, pp. 100–109.

[31] J. L. Singleton and G. T. Leavens, "Verily: a web framework for creating more reasonable web applications," in *Companion Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 560–563.

[32] S. Hallé, T. Ettema, C. Bunch, and T. Bultan, "Eliminating navigation errors in web applications via model checking and runtime enforcement of navigation state machines," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*. ACM, 2010, pp. 235–244.

[33] J. Nijjar and T. Bultan, "Bounded verification of Ruby on Rails data models," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2011, pp. 67–77.

[34] R. Morales-Chaparro, M. Linaje, J. Preciado, and F. Sánchez-Figueroa, "MVC web design patterns and rich internet applications," *Proceedings of the Jornadas de Ingenieria del Software y Bases de Datos*, pp. 39–46, 2007.

[35] J. Wojciechowski, B. Sakowicz, K. Dura, and A. Napieralski, "MVC model, struts framework and file upload issues in web applications based on J2EE platform," in *Proceedings of the International Conference on Modern Problems of Radio Engineering, Telecommunications and Computer Science (TCSET)*. IEEE Computer Society, 2004, pp. 342–345.