FANTASIA: A Tool for Automatically Identifying Inconsistency in AngularJS MVC Applications

Md Rakib Hossain Misu
Institute of Information Technology
University of Dhaka
Dhaka, Bangladesh
email: bsse0516@iit.du.ac.bd

Kazi Sakib
Institute of Information Technology
University of Dhaka
Dhaka, Bangladesh
email: sakib@iit.du.ac.bd

***Abstract*—AngularJS is prone to inconsistency issues because of the abstract interactions between Document Object Model (DOM) and JavaScript. It creates hidden bugs, and leads the application to failure. It becomes acute when developers use custom AngularJS directives for increasing code reusability and maintainability. To resolve the inconsistency issues, a static code analysis based approach FANTASIA is proposed. FANTASIA first extracts Abstract Syntax Tree (AST) and DOM from the AngularJS application's MVC modules including its custom directives. By traversing AST and DOM, next it finds the defined identifiers along with the associated data types of those identifiers. Finally, the extracted identifiers and data types are mapped and compared using a string matching algorithm to determine the consistency across the application. To evaluate FANTASIA, 25 open source AngularJS applications are used where 15 applications contain only MVC modules and rest of the applications contain both MVC modules and custom directives. The experimental result shows that FANTASIA produces overall 97.63% recall and 100% precision to correctly detect inconsistency in those 15 applications similar to existing approach AUREBESH. Interestingly, when custom directives are present, FANTASIA outperforms with a significant increase of overall 96.66% recall and 100% precision comparing to 76.97% recall and 100% precision of AUREBESH.**

Keywords-JavaScript; MVC; Inconsistency; Static Analysis.

I. INTRODUCTION

AngularJS is a JavaScript-based MVC framework used for developing loosely coupled web applications, which are known as Single Page Applications (SPA) [1]. It provides developers the flexibility to separate business logic in several reusable modules and components, such as model, view, controller, directive, service, etc. However, AngularJS is still prone to inconsistency issues because of wrong interaction between DOM and JavaScript [2]. This wrong interaction occurs, as AngularJS facilitates the application development by abstracting the DOM API method call between the JavaScript and HTML code.

AngularJS depends on the use of identifiers to represent *model variable*(s) (*mv*) and *controller function*(s) (*cf*). To represent the functionality, the identifiers of *mv* and *cf* should be consistent in the view. Besides, in AngularJS, views consist of various built-in directives, such as *ng-if*, *ng-count* [3]. To use built-in directives, developers have to assign *mv* and *cf* to these directives with a defined form, such as *ng-if="{mv}"*, along with specific data types. For example, *ng-if* directive takes boolean type of *mv* and *cf*. So, *mv* and *cf* are used in *ng-if* directive should be boolean type. Since JavaScript is loosely typed dynamic programming language, the developers have to keep in mind that, values assigned to *mv* and returned by *cf*, should be consistent to their expected types. Inconsistencies among these identifiers and the types, can potentially incur significant loss in the functionality and performance. The reason is that, the major functionalities of an application rely on *mv* and *cf*.

AngularJS also supports the Do not Repeat Yourself (DRY) feature [4]. It allows developers to create one directive and reuse it anywhere within the entire application. Despite having a lot of built-in directives, it also encourages the developers to create custom directives to enhance the re-usability of the code. Every custom directive has some specific properties that define its own view, model and controller. Sometimes, it is also used inside a view under a specific controller by following a parent child relationship. While using custom directives, inconsistency may arise not only within its own model, view and controller, but also between its parent view and controller. Unfortunately, developers do not get exceptions and warnings when inconsistency issue occurs [5]. It becomes the worst to find inconsistencies when an application contains multiple models, views and controllers.

Since the usages of AngularJS MVC framework for client-end application development are fairly new, there are few papers addressing the inconsistency issues. Two state-of the art works, TypeDevil [6] and AUREBESH [7] are proposed to detect inconsistencies in JavaScript applications. TypeDevil is capable of detecting inconsistency only within the JavaScript source codes. It performs dynamic source code analysis to detect data type inconsistency. On the contrary, AUREBESH is also able to detect inconsistencies in JavaScript MVC applications by performing static code analysis on both the JavaScript and HTML code. However, these approaches are not able to accurately identify inconsistencies in AngularJS MVC applications. The reason is that TypeDevil only dynamically analyzes the JavaScript source code instead of analyzing both JavaScript and HTML source code. AUREBESH only performs static source code analysis in MVC modules and never analyzes the presence of custom directives in AngularJS applications.

To resolve the inconsistency issues in AngularJS MVC applications, a static code analysis based approach FANTASIA is proposed. It first extracts Abstract Syntax Tree (AST) and DOM by performing static code analysis among the modules, such as model, view, controller and custom directive including its associated files. Then using AST and DOM, it searches for the identifiers that are used to represent *mv* and *cf*. The data types of *mv* and return types of *cf* are also drawn by exploring the AST nodes and DOM elements. Finally, to determine the inconsistencies across application, identifiers and data types, extracted from model and controllers are compared to those identifiers and data types extracted from views.

In order to evaluate FANTASIA, 25 open source AngularJS applications are used. Among these 25 applications, 15 applications contain MVC modules and rest of the 10 applications contain both MVC modules and custom directives. From experimental result analysis, it is observed that FANTASIA performs accurate with overall 97.63% recall and 100% precision to find inconsistencies in 15 applications with MVC modules, comparing to the existing approach AUREBESH [7]. When custom directives are present, FANTASIA achieves a significant overall 96.66% recall and 100% precision to identify inconsistencies in 10 AngularJS MVC applications that contain inconsistencies within both the MVC modules and custom directives.

The remainder of this paper is structured as follows. Section II describes the proposed approach for inconsistency detection with a concise description of each step. Implementation, evaluation and result analysis are discussed in Section III. Section IV deals with the existing techniques for fault and inconsistency detection in JavaScript applications. Finally, Section V concludes the paper by summarizing the contribution and possible future direction of this work.

II. Proposed Approach

To resolve the inconsistency issues, a static code analysis based approach FANTASIA is proposed. An overview of the proposed approach is depicted in Figure 1 as a block diagram. From the block diagram, it is seen that there are 9 modules, such as *MVC Components and Directive Identifier (MCDI)*, *DOMExtractor (DMEx)*, *DirectiveExtractor (DEx)*, *ASTExtractor (AEx)*, *ViewExtractor (VEx)*, *ModelExtractor (MEx)*, *ControllerExtractor (CEx)*, *MVC Group Builder (MGB)* and *Inconsistency Detector (InD)* that work collaboratively in several phases. TABLE I also represents the terms used for describing the proposed approach. The modules are depended on each other for taking input, providing output and giving feedback. A brief description of each of the modules is discussed in the following subsections.

A. MVC Components and Directive Identifier (MCDI)

MCDI module identifies and filters the MVC component source files (except library files) based on the file names and

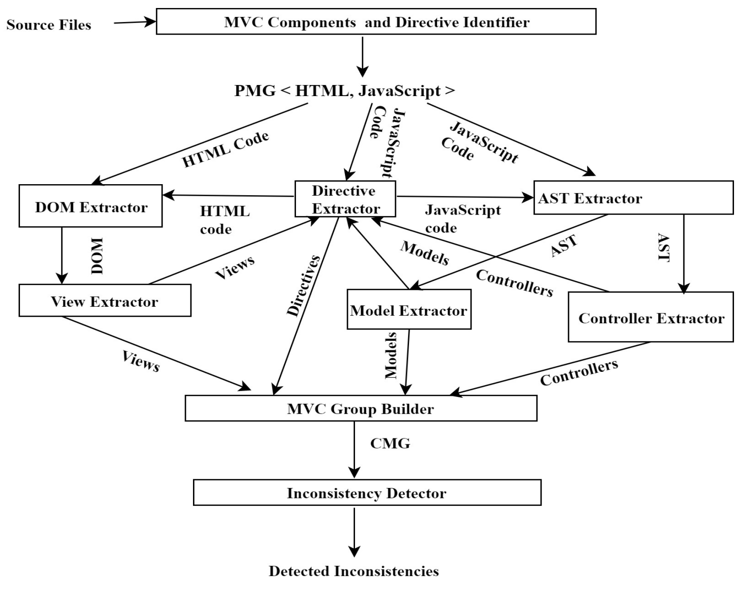

Figure 1. Block diagram of proposed approach.

module types. It is assumed that every module is written in a single source file and the file name should be self descriptive to figure which and what type of AngularJS module it is. After filtering, the module makes a list of directive definition files and extracts the application configuration file which is responsible for defining the routes and the correspondent views and controllers related to that routes. It provides the view and controller file names that are related and responsible for each route of the application. Using this information, a *Primary MVC Group (PMG)* is made that contains a list of HTML code for view file and JavaScript code for controller file in a form of tuple <view HTML code, controller JavaScript code>. Further, this group is used by the *DMEx* and *AEx* module and the list of directive files is used by the *DEx* module.

B. DOM Extractor (DMEx)

DMEx module uses HTML code as input from *PMG* that is provided by *MCDI* module. It also gets HTML source code from the *DEx* module (shown in Figure 1). It is responsible for transforming the HTML code into its DOM representations, which is used for analyzing the HTML elements and attributes. This module provides the extracted DOM to the *VEx* module for further extracting the AngularJS built-in directives and elements.

C. AST Extractor (AEx)

Similar to *DEx*, *AEx* module gets the input from *PMG*. It also gets JavaScript code from *DEx* module. The responsibility of *AEx* is to transform the JavaScript code into its AST representation. It provides the AST to the *ME* and *CEx* module. Those modules analyze the AST for further extracting *mv* and *cf*.

D. View Extractor (VEx)

VEx module analyzes the DOM and produces a set of *View (V)* objects. It extracts all the identifiers of *mv* and *cf* that are

TABLE I. LIST OF TERMS USED IN PROPOSED APPROACH

Term	Descriptions	Term	Descriptions
MCDI	MVC Components and Directive Identifier	*DMEx*	DOM Extractor
AEx	AST Extractor	*VEx*	View Extractor
MEx	Model Extractor	*CEx*	Controller Extractor
DEx	Directive Extractor	*MGB*	MVC Group Builder
InD	Inconsistency Detector	*M*	Model Object
V	View Object	*C*	Controller Object
CD-M	Model Object for Custom Directives	*CD-V*	View Object for Custom Directives
CD-C	Controller Object for Custom Directives	*PMG*	Primary MVC Group
UMG	Updated MVC Group	*CMG*	Complete MVC Group

used in the view. Generally, *mv* and *cf* are appeared as DOM elements or attribute value of AngularJS built-in directives. The attributes of AngularJS built-in directives accept a specific type of value. So, the accepted type of these built-in directives are also analyzed. Besides, *VEx* finds the presence of custom directives that are used in the DOM and makes a list of custom directives.

E. Model Extractor (MEx)

MEx takes AST as input from *AEx* module. It analyzes the AST of controller files and produces a set of *Model (M)* objects. It finds all the *mv* that are defined in the controller. The *mv* which are used in the view are binded with a view model variable (generally it is represented by *$scope* or *vm*). To find the identifiers of *mv*, *MEx* looks for the left hand side of the assignment expression. So, the identifiers are found as the properties of view model variable. For getting the type of *mv*, *MEx* considers 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 StringLiteral node than the inferred type is String). If the right hand expression is too complex, the assigned type cannot be inferred. In this case, the type is considered as *complex* for that identifier.

F. Controller Extractor (CEx)

Similar to *MEx*, *CEx* also receives AST from *AEx* module and analyzes the AST of controller files and generates a set of *Controller (C)* objects. It extracts the *cf* identifiers following the same way described in *MEx*. However, to get the assigned types for *cf* identifiers, the return type of each *cf* is considered. Finally, the modules *VEx*, *MEx* and *CEx* generates the sets of *Models (M)*, *Views (V)* and *Controllers (C)* objects. Each element of the set of *M, V* and *C* is considered as a tuple of *UMG* in a form of <M,V,C> that is used by *MGB* module.

G. Directive Extractor (DEx)

DEx module gets a list of custom directive definition files from *MCDI* module. Using the definition files, it extracts directive type, related view and controller files that are responsible for representing the functionality of that directive. The view files are fed to the *DMEx* module to generate DOM. After that, DOM is similarly extracted by the *VEx* module to create a View object for that custom directive that is represented by *CD-V*. It contains a list of *mv* and *cf* identifiers and types used in the custom directive view. The controller file of that custom directive is fed to the *AEx* module that also generates AST. Next, this AST is extracted by the *MEx* module to produce a Model object for that custom directive represented by *CD-M*. The *MEx* extracts identifier and type of the *mv*. Similarly, *CEx* generates a Controller object *CD-C* for that custom directive. It extracts the identifier and return type of *cf*. Finally, *DEx* module builds a list of *Directive* objects that contains the related model, view and controller for each directive.

H. MVC Group Builder (MGB)

The module *MGB* receives *UMG* and gets a list of *Directive* objects from the module *DEx*. It is responsible for building *CMG* by adding some new tuple with *UMG*. For every element of *UMG*, each directive is analyzed from the list of *Directive* objects based on its type. At first, for each directive, an empty tuple of Model *M*, View *V* and Controller *C* object is initialized and the View object of the directive *CD-V* is assigned to it. The reason is that the directive has its own view and it cannot be inherited from the parent view. Next, for each directive the type of the directive is checked. The type of a directive is determined by its scope property. If the scope is false, it refers that this directive does not manipulate the *cf* and *mv* of its parent controller. It directly uses the *cf* and *mv* properties from its parent controller to its view. So, the *CD-M* and *CD-C* of that directive are directly assigned to the empty M and C. If the scope is true, it means that this directive can prototypically inherit and manipulate the *cf* and *mv* of its parent controller. So, the collection of *mv* and *cf* used in both directive and its parent controller are assigned to the M and C, respectively. When the scope is isolated, it means that the *mv* and *cf* of this directive are isolated from its parent controller. For such, the directive's *CD-M* and *CD-C* are assigned to the empty M and and C, respectively. Finally, new tuple of <M,V,C> are added to the *UMG* to form the *CMG*.

I. Inconsistency Detector (InD)

The module *InD* gets *CMG* from *MGB* and provides a list of inconsistency. It mainly compares all the *mv* and *cf* among the Model, Controller and View object of each tuple of *CMG*. It is performed to identify the potential inconsistencies that exist within each tuple. At first, it searches the inconsistencies related to *mv* by iterating every *mv* that is used in the view and controller. For all such *mv*s that are defined in the controller, their identifiers are checked to see whether these also exists and are defined to the corresponding view. The checking is done based on string comparison. If it does not exist it means either these *mv* are not used in the view or their identifiers are inconsistent. So, there exists an identifier inconsistency and it is included in the inconsistency list. However, if *mv* exists, next the data type of the *mv* is checked into the view and controller. If the data type of the *mv* is dissimilar, corresponding to the view and controller, it means that a type inconsistency is present that is also included in the inconsistency list. Following the same process, inconsistencies in the *cf* are identified. It is assumed that *mv* and *cf* with unknown and complex types are matched with all types.

III. Evaluation

This section deals with the evaluation of proposed approach. A brief description of each aspect of evaluation is described in the following subsection.

A. Implementation

Since Command Line Interface (CLI) tools are getting popular for client-end application development, FANTASIA [8] is implemented in the form of a CLI using JavaScript programming language on top of *Node.js* framework. It is available as an open source *Node.js* package that can easily be installed using *Node.js* package manager (*nmp*). For identifying inconsistencies, developers have to run a command called *find-incons* in the application's base directory using the command prompt. For each identified inconsistency, an error message is shown containing the inconsistency type, file name and the location of the code where the inconsistencies are occurred. Tool demonstration of FANTASIA is available in [8].

B. Experimental Dataset

In total, 25 AngularJS MVC applications are used. These are chosen from a list of MVC applications mentioned in AngularJS Git-Hub page [9]. The applications that were chosen here were also used to evaluate the existing approach AUREBESH. Based on the presence of inconsistencies in the custom directives, these applications are categorized into two classes. Among the 25 applications, 15 applications containing MVC modules are categorized as Class A and rest of the 10 applications containing both MVC modules along with custom directives are categorized as Class B.

C. Fault Injection Study

Similar to AUREBESH [7], the efficiency of FANTASIA was measured by performing a fault injection study on the experimental dataset. The injection was performed by initializing mutations in the applications. The mutations were initialized in a way so that these could create inconsistencies in those applications. Inconsistencies within the applications depend on consistency properties. According to Frolin et al. [7], JavaScript MVC applications are inconsistent if the applications do not satisfy one of the following four consistency properties.

1) The controller and view can only use *mv* that are defined in the model.
2) The view only uses *cf* that are defined in the controller.
3) The expected types of corresponding *mv* in the view match the assigned types in the model or controller.
4) The expected and returned types of corresponding *cf* match in the view and controller.

Based on the consistency properties, Frolin et al. introduced 10 types of mutations [7]. The description of each mutation type is represented in TABLE III. Among these types, every mutation type corresponds to a violation of the above 4 consistency properties. In this experiment, these 10 types of mutations (mentioned in TABLE III) were also used. At first, the mutations were injected into the source code of those applications. After that, FANTASIA was run on the mutated version of the applications and analyzed whether FANTASIA could identify the inconsistencies initialized by the mutations. If the inconsistencies were identified, the result of the injection was noted as successful, otherwise failed. Finally, the numbers of successful and failed detections were counted for measuring precision and recall. For comparative analysis, AUREBESH was also run on the mutated applications and counted the number of successful and failed detections.

The results of identifying mutation type represent how well FANTASIA can detect the violation of corresponding consistency properties. For this study, at least 4 injections were performed per mutation type that amounts 30 to 40 injections in each application. It was noted that some mutation types were not applicable for all applications. For example, it is not mandatory that all controllers use the model variables. For these types of specific case, some mutation types were not considered. As a result, there were less than 40 injections injected in some applications. The location of the mutated code was chosen arbitrarily only if that line of code was applicable for current mutation type.

D. Result

After running both FANTASIA and AURBESH on the mutated version of Class A dataset, recall and precision are calculated based on successful and failed detection. TABLE II shows the fault injection study results over the Class A dataset where TI refers to total injection, SD refers to successful detection and FD refers to failed detection. As TABLE II shows, FANTASIA is very accurate yielding to an overall recall of 97.63% with 100% precision and gets perfect recall in 11 out of 15 applications. From TABLE II, it is also observed that existing approach AUREBESH also performs accurately with an overall recall of 97.20% with 100% precision and gets perfect recall in 11 out of 15 applications. So, both FANTASIA and AUREBESH perform similarly for identifying inconsistencies in those applications which contain inconsistencies only in MVC modules and not in custom directives.

Again FANTASIA and AUREBESH both were run on mutated version of Class B dataset to calculate the recall and precision. TABLE IV shows the fault injection study results over the Class B dataset. Here, FANTASIA also performs accurately with an overall recall of 96.66% with 100% precision and gets perfect recall in 5 out of 10 applications. However, AUREBESH does not perform well with an overall recall of 76.97% and 100% precision with no perfect recall. The reason for AUREBESH's poor performances is that it never analyzes the presence of custom directives in those applications. So, it was unable to identify those inconsistencies which were occurred within the custom directives. On the other hand, FANTASIA analyzes the presence of custom directives and

TABLE II. COMPARATIVE RESULT BETWEEN FANTASIA AND AUREBESH ON CLASS A DATASET

Applications	**Application Category**	**Size (LOC)**	**TI**	**FANTASIA**				**AUREBESH**			
				SD	**FD**	**Recall(%)**	**Precision(%)**	**SD**	**FD**	**Recall(%)**	**Precision(%)**
Angular Tunes	Music Player	185	35	35	0	100.00	100.00	35	0	100.00	100.00
Balance Projector	Finance Tracker	511	40	34	6	85.00	100.00	33	7	82.05	100.00
Cafe Townsend	Employee Tracker	452	40	40	0	100.00	100.00	40	0	100.00	100.00
Cryptography	Encoder	523	40	40	0	100.00	100.00	40	0	100.00	100.00
Dematerializer	Blogging	379	40	36	4	90.00	100.00	37	3	92.05	100.00
Dustr	Template Compiler	493	40	40	0	100.00	100.00	40	0	100.00	100.00
ETuneBook	Music Manager	5042	40	40	0	100.00	100.00	40	0	100.00	100.00
Flat Todo	Todo Organizer	255	40	40	0	100.00	100.00	40	0	100.00	100.00
GQB	Graph Traversal	1170	40	38	2	95.00	100.00	37	3	92.05	100.00
Hackynote	Slide Maker	236	40	40	0	100.00	100.00	40	0	100.00	100.00
Kodigon	Encoder	948	40	40	0	100.00	100.00	40	0	100.00	100.00
Memory Games	Puzzle	181	37	35	2	94.59	100.00	34	3	91.89	100.00
Shortkeys	Shortcut Maker	407	40	40	0	100.00	100.00	40	0	100.00	100.00
Sliding Puzzle	Puzzle	608	34	34	0	100.00	100.00	34	0	100.00	100.00
TwitterSearch	Search	357	40	40	0	100.00	100.00	40	0	100.00	100.00
Overall		**11747**	**626**	**612**	**14**	**97.63**	**100.00**	**610**	**16**	**97.20**	**100.00**

TABLE III. TYPES OF INJECTED FAULTS

No	Description	Property
1	Change the name of a *mv* used in line N of a view	1
2	Change the name of a *mv* used in line N of a controller	1
3	For a particular *mv* used in line N of a view, remove the declaration of that *mv* in a corresponding model	1
4	For a particular *mv* used in line N of a controller, remove the declaration of that *mv* in a corresponding model	1
5	Change the name of a *cf* used in line N of a view	2
6	For a particular *cf* used in line N of a view, remove the declaration of that *cf* in a corresponding controller	2
7	For a particular *mv* used in the view that expects a certain type T1, change the declaration of that *mv* in line N of a corresponding model so that the type is changed to T2	3
8	For a particular *mv* used in the view that expects a certain type T1 and declared in line N of a corresponding model, change the expected type to T2 by mutating the *ng* attribute name	3
9	For a particular *cf* used in the view that expects a certain type T1, change the return value of that *cf* in line N of the controller to a value of type 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, change the expected type to T2 by mutating the *ng* attribute name	4

able to identify the inconsistencies that are occurred within the custom directives.

From TABLE II and TABLE IV, it is observed that both FANTASIA and AURBESH attain 100% precision. The reason is that there is no occurrence of getting false positive results for successful and failed inconsistency identification. As, it is the assumption that all applications are developed by following proper coding convention, it prevents both approaches from getting false positive results.

IV. Related Work

Inconsistency occurs in JavaScript applications because of wrong interaction between DOM and JavaScript code. As a result, DOM-related faults and errors are partially responsible for inconsistency issues. However, AngularJS has gradually been developed over the last couple of years. Therefore, it is considered to be a new area of research. A few works have been found that directly discusses inconsistency issues in AngularJS MVC applications. Among those works, several studies [2][5][10][11] rigorously discuss DOM-related faults and errors that occur in JavaScript applications. Moreover, two recent studies [6][7] have addressed the inconsistency issues in JavaScript application development. So, considering all of those works, the knowledge domain is classified in two categories, such as *DOM Related Fault in JavaScript* and *Inconsistency in JavaScript*. A brief description of each category is mentioned in the following subsections.

A. DOM Related Fault in JavaScript

Since AngularJS MVC framework contains both HTML DOM and JavaScripts, DOM related errors and faults are directly responsible for inconsistency issues. Several studies have been conducted to analyze the behavior of DOM in JavaScript applications, such as Forlin et al. performed an empirical study [10] for identifying numerous errors and faults in JavaScript based web applications. Both static and dynamic source code analysis are performed in this study that has identified different characteristics of JavaScripts faults. Further, these characteristics of JavaScript faults are used by Ocaizer et al. [11] to identify errors and faults during JavaScript application development. They proposed an automatic fault localization approach AUTOFLOX by analyzing the JavaScript fault characteristics. By performing dynamic backward program slicing, AUTOFLOX can localize faults within the JavaScript based web applications. Besides, the evaluation result of AUTOFLOX shows that about 79% of reported JavaScript errors and faults are DOM related [11].

On the other hand, based on those results, Forlin et al. [2] observed that almost 65% of JavaScript faults are DOM related faults that occur because of the wrong interaction of JavaScript code and DOM element using incorrect identifier. However, in development phase, these located faults are needed to be resolved. In order to resolve these faults, an automatic fault repairing technique VEJOVIS was proposed by Forlin et al. [5]. This technique includes the combination of both static and dynamic code analysis with backward program slicing. The outcome of this technique is the categorization of some

TABLE IV. COMPARATIVE RESULT BETWEEN FANTASIA AND AUREBESH ON CLASS B DATASET

Applications	Application Category	Size (LOC)	TI	FANTASIA				AUREBESH			
				SD	FD	Recall(%)	Precision(%)	SD	FD	Recall(%)	Precision(%)
Angular Qize	Quiz Maker	523	36	36	0	100.00	100.00	30	6	83.33	100.00
Angular Table	Component	327	35	35	0	100.00	100.00	30	5	85.71	100.00
Angular Ui-Grid	Component	243	35	35	0	100.00	100.00	26	9	74.78	100.00
C3-Chart	Librery	763	35	35	0	100.00	100.00	27	8	77.14	100.00
Color Chooser	Component	134	30	30	0	100.00	100.00	23	7	76.66	100.00
Date Picker	Component	278	40	37	3	92.20	100.00	30	10	75.00	100.00
Directives Lab	Directive Example	412	40	39	1	97.50	100.00	31	9	77.50	100.00
GemStore2	Game	453	40	38	2	95.00	100.00	28	12	70.00	100.00
Responsive Slider	UI Design	359	37	33	4	89.18	100.00	25	12	67.56	100.00
Text Editor	Editor	192	40	37	3	92.50	100.00	33	7	82.50	100.00
Overall		**3684**	**368**	**355**	**13**	**96.66**	**100.00**	**283**	**85**	**76.97**	**100.00**

common types of faults. However, FANTASIA completely differs from these works, as in these works non-MVC applications and frameworks are considered. These applications have various architectural patterns compared to AngularJS MVC framework. So, these works are not compatible to resolve inconsistency issues in AngularJS MVC applications.

B. Inconsistency in JavaScript

Since, JavaScript is a dynamic programming language, it does not provide compile-time warning if a program contains identifier or data type inconsistencies [11]. Both of these inconsistencies are responsible of creating hidden bugs and failures. However, in a survey study, it is found that among 460 developers, 39% of those consider that silent failures caused by identifier or type inconsistencies, are real problems during application development [12]. For identifying type inconsistencies, Michael et al. proposed an approach called TypeDevil [6] that can detect type inconsistencies by performing dynamic analysis on JavaScript code. To evaluate the approach, TypeDevil [6] was applied on JavaScript code collected from various applications. The evaluation shows that it can detect type consistency within the JavaScript files.

In order to detect inconsistency, an approach AUREBESH [7] was proposed that can detect both the type and identifier consistencies by performing static code analysis in JavaScript MVC applications. To evaluate AUREBESH, a fault injection study was conducted on 20 open source AngularJS applications considering to be representative of MVC applications. The result of this study shows that AUREBESH can detect inconsistencies and some real world bugs in those applications.

However, TypeDevil [6] cannot find inconsistency in AngularJS MVC applications since to find type and identifier inconsistencies in MVC applications, both the controller JavaScript and view HTML code should be analyzed. TypeDevil [6] does not analyze the inconsistencies between the HTML and JavaScript code rather it only analyzes the JavaScript code. FANTASIA resolves this problem by performing static analysis on both HTML and JavaScript code instead of performing dynamic analysis only within JavaScript code. On the other hand, while using custom directives in AngularJS applications, AUREBESH [7] cannot detect inconsistencies because it does not analyze the presence of custom directives. FANTASIA also differs from AUREBESH as it considers the presence of custom directives across the applications and identifies those inconsistencies that occur within the custom directives and MVC modules.

V. CONCLUSION AND FUTURE WORK

The presence of inconsistencies (e.g., identifier and type inconsistency) in AngularJS applications produces hidden bugs, which reduce the maintainability and readability of code. During development, it is hard to identify inconsistencies since JavaScript does not provide compile time warn if any inconsistency occurs. Detecting inconsistency becomes more difficult when developers use custom directives with MVC modules. However, existing approach can only identify inconsistencies that occur in MVC modules omitting the presence of custom directives. To resolve this issue, FANTASIA is proposed that performs static code analysis across the application and analyzed the presence of custom directives to detect inconsistencies.

According to the result analysis, FANTASIA achieves an overall 97.63% recall and 100% precision similar to existing approach AUREBESH, to detect inconsistency in 15 applications that contain inconsistency in MVC modules. Comparing to AUREBESH, it outperforms with an overall 96.66% recall and 100% precision to detect inconsistency in 10 applications containing inconsistency both in MVC modules and custom directives. The reason for FANTASIA's significant increase of recall is analysis the presence of custom directives.

Incorporating FANTASIA with the existing tool AUREBESH, to detect inconsistency in other JavaScript MVC frameworks (e.g., *Ember.js*), can be a future research scope. As FANTASIA is only applicable to the primary version of AngularJS framework, the future work is to make FANTASIA compatible to the latest versions of AngularJS. Future scope also includes to make FANTASIA compatible for TypeScript or CoffeScript based MVC applications. Currently, the scope of this proposed technique is to identify inconsistency only in AngularJS MVC applications developed in JavaScript. However, in future this approach can be further used to automatically remove and fix inconsistency in MVC applications.

REFERENCES

[1] V. Balasubramanee, C. Wimalasena, R. Singh, and M. Pierce, "Twitter bootstrap and angularjs: Frontend frameworks to expedite science gateway development," in *2013 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2013, pp. 1–1.

[2] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah, “An empirical study of client-side javascript bugs,” in *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2013, pp. 55–64.

[3] Angular, “Ng-if directive,” https://docs.angularjs.org/api/ng/directive/ngIf, 2017, [Online], [Accessed 2017-06-15].

[4] AngularJS, “AngularJS.org,” https://angularjs.org/, 2017, [Online], [Accessed 2017-06-15].

[5] F. S. Ocariza Jr, K. Pattabiraman, and A. Mesbah, “Vejovis: suggesting fixes for javascript faults,” in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 837–847.

[6] M. Pradel, P. Schuh, and K. Sen, “Typedevil: Dynamic type inconsistency analysis for javascript,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 314–324.

[7] F. S. Ocariza Jr, K. Pattabiraman, and A. Mesbah, “Detecting inconsistencies in javascript mvc applications,” in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 325–335.

[8] FANTASIA, “FANTASIA,” https://www.npmjs.com/package/fantasia-inconsistency-detector, 2017, [Online], [Accessed 2017-06-15].

[9] GitHub, “GitHub/AngularJS,” https://github.com/angular/angular.js, 2017, [Online], [Accessed 2017-06-15].

[10] F. S. Ocariza Jr, K. Pattabiraman, and B. Zorn, “Javascript errors in the wild: An empirical study,” in *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*. IEEE, 2011, pp. 100–109.

[11] F. S. Ocariza Jr, K. Pattabiraman, and A. Mesbah, “Autoflox: An automatic fault localizer for client-side javascript,” in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 2012, pp. 31–40.

[12] M. Ramos, M. T. Valente, R. Terra, and G. Santos, “Angularjs in the wild: a survey with 460 developers,” *arXiv preprint arXiv:1608.02012*, 2016.