TryOpenJML - A Verily based web application for learning about the Java Modeling Language

2016

Tushar Deshpande
University of Central Florida

TRYOPENJML – A VERILY BASED WEB APPLICATION FOR LEARNING ABOUT THE JAVA MODELING LANGUAGE

by

TUSHAR DESHPANDE
B.E. Visveswaraya Technological University - Belgaum, 2012

A thesis submitted in partial fulfilment of the requirements
for the degree of Master of Science
in the Department of Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term
2016

Major Professor: Gary T. Leavens

ABSTRACT

This thesis has a two-fold purpose. On the one hand, the web applications are an important part of life. On a day to day basis, from managing our heath care choices to banking, to connecting to a friend, almost everything is done through a web application. Development of these applications is also a very trend-driven domain. Numerous web frameworks are available today, but almost none has been created taking reliability into consideration. With the combination of application construction recipes and static analysis, the Verily framework was created to build more reliable web applications.

On the other hand, the goal of the Java Modeling Language (JML) has to be conveyed to the world. It is a language that can go hand in hand with existing code, having a wide range of tools that help build practical and effective designs. There are many tools available for JML: jmldoc for web pages, jmlunit for unit tests, jmlc for class files, etc. I will be using the tools for Runtime Assertion Checking (RAC) and Extended Static Checking (ESC). These checks warn about the possible runtime exceptions and assertion violations. The benefits of JML assert statements over Java assertions are that they support all JML features.

The question that I am concerned with, in this thesis, is how the Verily Framework can contribute to the domain of web application development. Keeping this question in mind, my objective is to create a tutorial which will aid in learning about JML. The tutorial will let the potential users read and write JML specifications and use JML tools, explain basic JML semantics, and let them know where to go for help if they need more details.

This thesis work is dedicated to my family, who have always loved me unconditionally and whose good examples have taught me to work hard for the things that I aspire to achieve.

ACKNOWLEDGMENTS

Firstly, I would like to extend most profound appreciation to my advisor, Dr. Gary T. Leavens, for his encouragement and excellent guidance, and additionally to introduce me to Java Modeling Language. Without his support, this thesis would not have been possible. I would also like to thank the members of my thesis committee, Professors Damla Turgut and Damian Dechev for their advice and guidance during the entire process.

I would also like to thank John Singleton for the stimulating discussions, valuable suggestions and guidance. Special thanks to Shraddha who has been a constant source of support and encouragement during the challenges of graduate school and life.

TABLE OF CONTENTS

LIST OF FIGURES x

CHAPTER 1: INTRODUCTION 1

1.1 Why the Verily Framework ? 2

1.2 Why create a tutorial for Java Modeling Language ? 3

CHAPTER 2: LITERATURE REVIEW 4

2.1 Introduction 4

2.2 Prior tutorials on JML 4

2.3 Reliability in software 5

2.4 Containters and Docker 7

CHAPTER 3: TUTORIAL DESIGN 8

3.1 Web-based Application 8

3.1.1 Background of the framework 8

3.1.2 Design pattern 9

3.1.3 Javascript Bridge 12

3.1.4 Front-End 13
3.1.4.1 AngularJS 13
3.1.4.2 Freemarker 14
3.1.4.3 Bootstrap CSS 16
3.1.5 Back-end 17
3.1.5.1 Background on OpenJML API 17
3.1.5.2 Docker Containers 18
3.1.5.3 NGINX 20
3.2 Tutorial 20

CHAPTER 4: IMPLEMENTATION 21
4.1 Preview 21
4.2 OpenJML Runner 21
4.2.1 Runtime Assertion Check (RAC) 22
4.2.2 Extended Static Check (ESC) 23
4.3 Performance evaluation of the Verily framework 24
4.4 Tutorials 25
4.4.1 Overview of OpenJML 25

4.4.1.1 Assertion 25

4.4.1.2 Using TryOpenJML 26

4.4.2 Writing your first pre-condition 26

4.4.3 Writing your first post-condition 27

4.4.4 How do JML annotations work? 27

4.4.5 Model and Ghost fields 28

4.4.5.1 Model 28

4.4.5.2 Ghost 29

4.4.6 Modifiers and Assignable clauses 29

4.4.6.1 Modifiers 29

4.4.6.2 Assignable clauses 30

4.4.7 Invariants 30

CHAPTER 5: FINDINGS 32

5.1 Web-based Application 32

5.2 Tutorials 34

CHAPTER 6: FUTURE DIRECTIONS 35

CHAPTER 7: CONCLUSION 36

7.1 The Verily Framework 36

7.2 Tutorial on Java Modeling Language 36

7.3 TryOpenJML - the web-based application for learning about JML 37

APPENDIX : SOURCE CODE 38

LIST OF REFERENCES 55

LIST OF FIGURES

Figure 3.1: MRR design overview diagram 8

Figure 3.2: Example Method - *RuntimeAssertionChecker.java* [1] 10

Figure 3.3: Example Router - *RuntimeAssertionChecker.java* [1] 11

Figure 3.4: Working of FreeMarker Templates [2] 15

Figure 4.1: TryOpenJML web application preview 21

Figure 4.2: Runtime Assertion Check - Sequence Diagram 22

Figure 4.3: Extended Static Check - Sequence Diagram 23

Figure 4.4: Test configuration for Verily based application under test [3] 24

Figure 4.5: Test configuration for Node.js based application under test [3] 24

Figure 5.1: Test Throughput of Verily v/s Non-verily [3] 32

CHAPTER 1: INTRODUCTION

Over the previous decade, the web has been embraced by a huge number of organizations and personnel as an economic channel to impart and trade data with prospects and transactions with clients and users. Nowadays, the web is much more than the display of text and graphics. There are web pages that allow dynamic personalized content to be pulled down by users according to their likes and settings. Moreover, client-side scripts can be run within internet browsers to make them look and act unique to each application. For example, Maps, and Email apps that load in the same browser look and act very differently. It is possible to create apps like this by adding app-specific code to generic functionality, using a framework.

Frameworks are not necessary for the development of an application. But they make the development process better by giving it a structure that makes the application upgradable and maintainable. Apart from that, the need for a framework arises when we want to create a layer of abstraction for the user-facing design template in the front end of an application and also for the base code related to computing answers for HTTP requests and responses. The level of abstraction depends on the choice of framework. Such web applications work in a similar pattern: receiving HTTP requests, sending the code to generate frontend web pages, and creating a response with the generated content.

There have been an enormous number of frameworks introduced to build web applications. These include PHP based frameworks like Zend, Laravel, and CakePHP, Python based ones like Django, Flask, and Pyramid, and Javascript based ones like AngularJS, Meteor and the React Framework. This list will go on and on with hybridization of frameworks to get the best from what is available.

The need for this level of abstraction comes into the picture as websites capture, process, store, and transmit sensitive customer data (e.g., personal details, credit card numbers, social security

information, etc.) for immediate and prolonged use. A framework that enhances the reliability of web applications should accomplish this task. The Verily framework may be a viable option for enhancing reliability; it features "recipes" (patterns) for handling common problems, such as global mutable state, in ways that makes the program more easily verified.

The problem I propose to solve is to build a web application to test and evaluate Verily and its features. The web application, TryOpenJML, will also educate its users about Java Modeling Language (JML).

1.1 Why the Verily Framework ?

There has been significant work done in enhancing web frameworks keeping in mind concepts like DRY (Don't Repeat Yourself), convention over configuration and scaffolding, with a broad focus on issues like performance and productivity. However, very little has been done by the open source world to improve the reliability of web applications. Hence, John Singleton and Gary Leavens created a web-based framework, Verily [4], which promises to improve the reliability aspect of the requests on the web.

Verily is a web framework for Java that supports the development of verifiable web applications. Rather than verification during *a posteriori* analysis, Verily checks for correctness of specifications statically with the help of pre-constructed individual recipes. A performance evaluation of the reliability aspect of the Verily framework built with AngularJS as front-end and Verily as the back-end to a Javascript framework built with AngularJS as front-end and Node.js as the back-end is presented in chapter 4.

1.2 Why create a tutorial for Java Modeling Language ?

The Java Modeling Language (JML) is the *lingua franca* of researchers working on specification and verification techniques and tools for Java [5]. More than twenty groups have worked on development and upgrading various features of JML. They have built a plethora of tools for verification and automated validation. This web tutorial will cover the initial know-how required to understand the present JML features specifying the functional behavior of sequential Java classes. Users will get firsthand experience of writing JML specifications like pre- and post-conditions, invariants, constraints, and ghost and model fields.

The rest of this thesis is organized as follows. In Chapter 2 there is a Literature Review, which discusses all the resources that have been referred to while building the application as well as creating the tutorials. Later in Chapter 3 we will be going through the design and development of the application and the supporting tutorial for JML. Similarly, in Chapter 4 findings of both app as well as tutorials will be discussed. Finally, Chapter 5 and Chapter 6 describes about the Conclusion and Future Work related to JML and later the Appendix will consist of the Source Code.

CHAPTER 2: LITERATURE REVIEW

This chapter covers the background research leading up to the creation of TryOpenJML and reviews of other related materials which give a precise idea about OpenJML, the Verily Framework and reliability of web-based software application in the present scenario.

2.1 Introduction

Singleton and Leavens [4] have introduced The Verily Framework, a web-based framework compatible with most versions of Java, that will aid in development of web-based applications that are easily verifiable. Other than this, they have also initiated use of two Recipies, the *Core Recipe*, an architecture which is a viable replacement for traditional Model View Controller, and the *Global Mutable State Recipe*, which aids in the use of sessions inside an application without placing it in the global mutable state.

Apart from this, Singleton and Leavens have introduced a *JavaScript Bridge*, which automatically exports Methods written in Java for Verily to JavaScript. In this thesis we have implemented TryOpenJML as a Verily application using this *JavaScript Bridge* to AngularJS.

2.2 Prior tutorials on JML

Leavens, Kiniry, and Poll [6] presented a tutorial on JML, which covered the various aspects of JML. It described JML's vast tool-set for automated static as well as dynamic checking which include an extensive tool set for verification and automated checking. It mainly consisted of the usage of data types, including pre-conditions, post-conditions, frames, invariants, history constraints,

ghost and model fields, and specification inheritance in JML.

Leavens [5] presented a tutorial which will introduce JML features that are useful for specifying the functional behavior of a sequential Java class and interface. The attendees also got a hands-on experience writing JML specifications for data types, including pre-conditions, post-conditions, frames, invariants, constraints, model and ghost fields, and specification inheritance. They saw how to verify object-oriented code using supertype abstraction for modular reasoning. Towards the end, there was an exchange of ideas on improving existing JML tools, open research problems, and future directions for research related to JML, including ways to connect JML to various theorem provers.

David Cok [7] has introduced OpenJML, which uses the OpenJDK compiler to enhance the processing of source files to add appropriate checks, which assertions and other specifications hold during execution of the program. OpenJML can be used with Eclipse as a plugin, in command-line interface tools and programmatically through an API. Abstract Syntax Trees, type information, compilations and checking commands, and the results of verification attempts can be accessed through this Application Programming Interface (API), to get good coverage in unit test cases.

The tutorials presented by Leavens, Kiniry, and Poll [6] and Leavens [5] with OpenJML [7] used as an API together are the inspiration behind TryOpenJML application.

2.3 Reliability in software

Tian and Li Ma [8] have aimed to improve web software reliability by reducing web problems closely identified with web source contents and navigation. Using information about web accesses and related failures extracted from existing web server logs, we build testing models that focus on the high-usage, high-leverage subsets of web pages for efficient problem detection and reliability

improvement. The authors have also extended their approach to address testing, development issues for the always evolving web, and reliability defect analysis as a whole by analyzing the dynamic web contents and other information sources which are not covered in their case study in this paper.

Tian and Li Ma [9] have analyzed web server error logs and the corresponding referral pairs from web access logs to identify typical web errors and also characterized them for a taxonomic purpose. As a result, they have identified missing files to be the primary type of web error and have classified them according to their incoming referral links into internal, external, and user errors. Apart from that, they have also found different quality assurance initiatives to deal with various types of web problems which prove useful for improvement of web reliability.

Jiang, Xu, Dong, Jin, and Liao in [10] have found Web-services based application to be most used among all. Hence, they have implemented a framework by extending and enhancing the Spring Security framework. It improves the security level of systems and does not affect the speed of data. The writers of this article have mixed RSA and DES and used it in the Spring Security's ACL (access control list) mechanism. The whole framework was tested on VeePalms, which proved the effectiveness of this security framework. The Verily Framework on the other hand is built to focus on the reliability aspect and, in future will be tested on similar models as VeePalms.

Loadster [3] is a cloud-hybrid performance testing solution consisting of Loadster Workbench, which runs as an application on your Mac or Windows system and connects to Loadster Engine and Loadster Cloud. Loadster Engine is a local agent working on our system, and Loadster Cloud is an agent working on cloud simulating more than a thousand users when needed. We will be using free version of this solution which gives 25 virtual users in the Loadster Engine during performance evaluation of the Verily framework with Node.js framework for testing of web services, web applications, and static websites.

2.4 Containters and Docker

Ivan Amelia *et. al.* from *Cisco Systems, Inc.* and Lars Herrmann *et. al.* from *Red Hat Software* in a white-paper [11] have explained how Linux containers and Docker are poised to change radically the way applications are built, shipped, deployed, and instantiated. The authors also claim that Dockers accelerate application delivery by packaging applications along with their dependencies. As a result, these containerized applications can operate in various development, test, and production environments. The platform on which these environment are created, can be a physical server, virtual server, public cloud, or even a network device.

Brandon Chavis and Thomas Jones in a whitepaper for *Amazon Web Services (AWS)* [12] say that, AWS is a natural complement to Linux containers because of the wide range of scalable infrastructure services upon which containers are deployed. The *AWS Elastic Beanstalk* includes support for Docker containers and Amazon Elastic Cloud 2 Container Service (Amazon ECS). Amazon ECS was designed from the ground up to manage the Docker containers at various scales. It is built upon several important features, such as cluster management and support for multiple container schedulers, to facilitate and orchestrate large-scale deployments of containers across managed clusters of Amazon EC2 instances.

The above chapter gives a brief account of most of the pieces of literature that were reviewed while writing this thesis. There was more information that was present on the Internet, which is cited in the references.

CHAPTER 3: TUTORIAL DESIGN

This chapter is bifurcated into a section about the design of the web-based application and a section on the design of the verbatim in the tutorial.

3.1 Web-based Application

3.1.1 Background of the framework

In Verily, Singleton and Leavens [4], have introduced two recipes, viz. the Core Recipe and the Global Mutable State Recipe. The Core Recipe is an architecture for web apps designed to replace traditional server-side Model View Controller design pattern with the *Method Router Response* pattern.

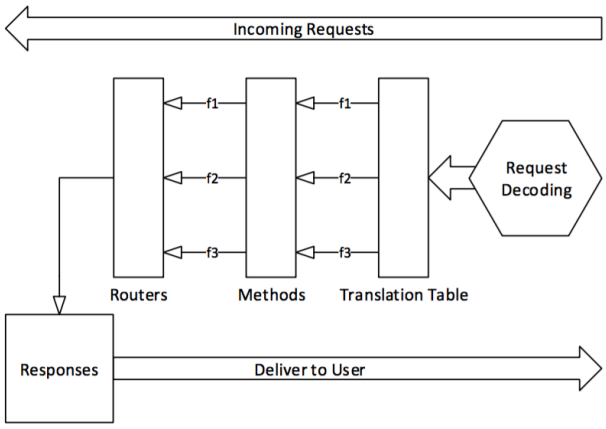

Figure 3.1: MRR design overview diagram

MRR's design has three kinds of components: Methods process application requests, Routers direct control flow, and the content and results (e.g., HTML or JSON) are shown with the help of Responses [4]. The explanation for each of Method, Router and Response in detail in the next sub-section. Moreover, the Global Mutable State recipe is presented so that it replaces the traditional *session* variables because their utilization in an application increases model coupling. The aim of the Global Mutable State (GMS) recipe is to limit such detrimental effects of sessions while giving a means for conversational flow in an application. With the Core recipe and the GMS recipe, the Verily Framework also has two supplemental features: the JavaScript bridge, which is a characteristic of Verily that exports Methods to JavaScript as a matter, of course and enables client-side applications to call Methods as if they were defined in JavaScript. This compliments the use of Verily in Single Page Applications.

Many applications rely on client-side JavaScript framework to handle user interaction with the application. Usually, such applications are single page web apps.

3.1.2 Design pattern

In Verily, ***Methods*** are the functions that are responsible for resolving the logic of an application on front-end. They are member functions of a class with the type signature as following,

```
1 public static final void(...)
```

```java
package methods;

import lang.InvalidClassfileException;
import runners.CheckerResult;
import runners.RACRunner;
import verily.lang.*;

import java.io.IOException;

public class RuntimeAssertionChecker{

    public static final CheckerResult run(String Source){

        RACRunner runner = new RACRunner(Source);

        try {
            return runner.run();
        } catch (IOException | InterruptedException | InvalidClassfileException e) {
            return new CheckerResult(e.getMessage(), false);
        }

    }

    public static final void metadata(){}
    public static final void language(){}
}
```

Figure 3.2: Example Method - *RuntimeAssertionChecker.java* [1]

As web applications are delivered over HTTP in a plain text query string, much processing is done to transform these HTTP requests into program data. Verily Methods define an explicit guarantee for converting these requests into data and also to map such requests into executable code. These statically-checked guarantees are as follows:

Requests are ensured to be Well-Formed: A well-formed request is a request that follows all semantic rules like having a complete request header and body. The idea of ensuring well-formed requests is similar to that of Google Web Toolkit, but it does not require the client-side app to be written in JavaScript.

URI/Class Isomorphism: Methods are mapped onto the URI structure in such a way that, it is isomorphic to where the Methods are placed in classes. For example, if a Method *harnessMain* belongs to the class Main, then the URI that maps to the Main.harnessMain Method would be:

http://myapp/Main/harnessMain

Router Parity: Verily checks for a Router definition for each method that is defined. For this,

source files of classes must be saved in the Methods folder of the Verily Project.

Routers handle the control flow in an application. As indicated in Figure 1, although Methods and Routers are always present in pairs, they are not dependent on each other. This feature has an advantage, as it separates application and navigational logic. An example of Router is shown in Snippet 2.

```
1  package routers;
2
3  import runners.CheckerResult;
4  import runners.RACRunner;
5  import verily.lang.*;
6
7  import java.util.HashMap;
8
9  public class RuntimeAssertionChecker{
10
11     public static final Content run(String Source, CheckerResult result){
12
13         Content response = new JSONContent();
14
15         response.setContent(RACRunner.checkerResultToRise4RunFormat(result));
16
17         return response;
18     }
19
20     public static final Content metadata(){
21         Content response = new JSONContent();
22         response.setContent(new TemplateHTMLContent("RuntimeAssertionCheckerMetadata.ftl", new HashMap()).getContent());
23
24         return response;
25
26     }
27
28    public static final Content language(){
29        Content response = new JSONContent();
30        response.setContent(new TemplateHTMLContent("Language.ftl", new HashMap()).getContent());
31
32        return response;
33    }
34 }
```

Figure 3.3: Example Router - *RuntimeAssertionChecker.java* [1]

Like Methods, Routers have some conventions to follow, *viz*:

Method Parity for each Router: There must be a method with matching Method name and class name.

Request/Call Equivalence: The parameters of Routers and Methods must be matching in name and similar in type; but, Routers can have a return type declared for them. Declaring a return type for

Routers allows elimination of mismatched content type errors through static analysis. Note that, data always passes from Method to Routers and not the other way around.

No Mutation of Global State: Routers can read from global mutable state but not write into that state. The non-mutable Global State for Routers makes Routers easier to test. Similar to Methods, source files of classes must be saved in the Routers folder of each Verily Project.

Responses encode the way content is rendered by the browser. A Response must be a subclass of the Content class or use one of the many provided subclasses of the Content class. There are two reasons for encoding the rendering information in the Responses. Firstly, all domain-specific features will be included in Content Classes. This will create a layer of abstraction. Secondly, unit tests can directly verify data from the Responses, rather than creating mock browsers to do the verification. Verily uses the templating capabilities of *Freemarker* template engine, and this is supported by *TemplateHTMLContent class*.

3.1.3 Javascript Bridge

Verily can export Methods to JavaScript on its own. This enables client-side applications to call Methods as if they were defined in JavaScript natively. Singleton and Leavens [4] have shown the use of both the synchronous and asynchronous JavaScript Bridge functionality. The demonstration with Backbone.js, which is a JavaScript library, was given in the original conference paper [4], to show how Verily solves several problems related to bridging client-side JavaScript to server-side applications. Moreover, a similar initiative is taken in this thesis to implement the integration of Verily using its Javascript Bridge, translating into AngularJS framework for TryOpenJML application.

3.1.4 Front-End

The Front-end / user-facing side of the TryOpenJML application is built using AngularJS.

3.1.4.1 AngularJS

AngularJS is a structural framework used to create dynamic web applications. It uses HTML as a template language and lets the user extend HTML's syntax for clear and concise display of an app's components. AngularJS and HTML when used together can create dynamic web applications.

The transition between static pages and dynamic applications is done by introducing a library, where all the functions that are useful for creating an application are stored. Moreover, the framework is an implementation of a web application where the user's code fills in the details. Another approach taken by AngularJS is that it teaches the browser new syntax with the help of angular directives. Hence, making it a completely a client-side solution.

Everything a user needs to build a CRUD (Create, Read, Update, Delete) application in a cohesive set: reusable components, dependency injection, form validation, basic templating directives, and data-binding are made available in the library.

Why use AngularJS ?

The reason behind using AngularJS in building of the *TryOpenJML* application is multi-fold and is mentioned below:

Registering callbacks: Registering callbacks makes it hard to scan for trees in the forest. Eliminating common boilerplate code such as callbacks reduces a lot of JavaScript coding in the application.

Manipulating the HTML DOM programmatically: Manipulating the HTML DOM leads to clunkier, more error-prone code. Low-level DOM manipulation tasks can be avoided by declaratively de-

scribing how the UI changes as the state of the application changes. AngularJS, if used as per its documentation[13], will never try to directly manipulate the DOM.

Marshaling data to and from the UI: CRUD operations are in the majority of AJAX application tasks. From marshaling data from an internal object in a form to validating the form, displaying errors, reporting with internal code back to the server, such CRUD operations create much boilerplate code. AngularJS eliminates most of this code so that user can focus on implementation details rather that common code used across the application.

Writing tons of initialization code just to get started: Even to get a "Hello World!" printed in an AJAX app requires much initial code to get the services up and running. With Angular, there are pre-built services which are auto-injected in the application to work through the initial bootstrap process. Also, these services are editable, to give the programmer total control over the initialization process.

3.1.4.2 Freemarker

Apache FreeMarker [2] is a Java library for a template engine which generates HTML pages, emails, source code and configuration files depending on changing data and templates. These templates are written in FreeMarker Template Language(FTL), which is a specialized language but not a full-fledged programming language. An example binding between HTML and FTL is shown in the diagram below.

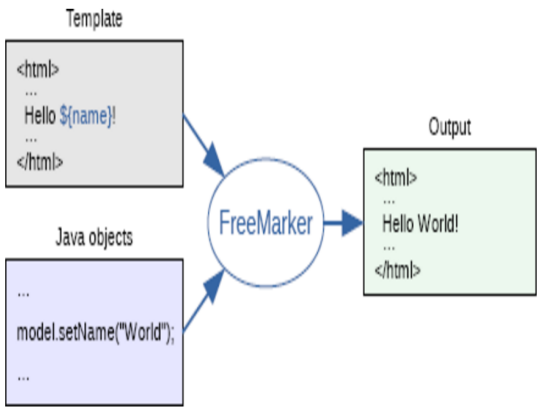

Figure 3.4: Working of FreeMarker Templates [2]

Although FreeMarker was initially created for generating HTML pages in MVC web applications, but it can also be used in non-MVC application such as *TryOpenJML*, which is based on Method Router Response pattern.

Why use FTL ?

FTL was used in the *TryOpenJML* application because:

Powerful template language: Equipped with iterators, conditional blocks, macros, functions, templates, string and arithmetic operations, and much more, FTL can be considered a powerful template language.

Multipurpose and lightweight: FTL does not have any dependencies, can give outputs in any format, there are many configuration options, and it is entirely pluggable; i.e, FTL can load templates from any place.

Internationalization/localization-aware: FTL is sensitive to number and date/time formatting as per location. Also, templates can be localized to a certain extent.

XML processing capabilities: FTL gives its user the power to process the XML DOMs into the data-model.

Versatile data-model: A tree of variables is used for exposing Java objects to the template through pluggable adapters.

3.1.4.3 Bootstrap CSS

Bootstrap is an open-source front-end library for building web applications and websites. It contains CSS and HTML based design templates for elements like forms, buttons, navigation and other interface components. Also, it has an optional JavaScript extension which can be used to give individual properties to elements. All the scripts related to these elements and Bootstrap libraries resides on client-side.[14]

Why use Bootstrap ?

The Bootstrap libraries are used in *TryOpenJML* application because:

Compatibility: Bootstrap is compatible with the latest versions of most of the well -known browsers and on all the key platforms.

Availability: Bootstrap is open-source and available on Github. Hence, it is easily accessible and contributed to, by many developers and designers around the globe.

Is supported by HTML5: Bootstrap is supported by HTML5 elements and CSS properties.

Grid implementation: A grid in Bootstrap is a network of imaginary horizontal and vertical lines that are uniformly placed to use as a reference to place web elements on a web-page. Bootstrap has a grid system that can be scaled up to 12 columns as needed for device or viewport size increases. This increases the responsiveness of user interface to change in size of browser. It has easy layout options, predefined classes for generating more semantic layouts. Containers are used to place elements inside the grid.

UI responsiveness: Fluid grid responsive system together with containers makes the User Interface responsive to changing screen size. Utility classes with the aid of media query are used for showing

and hiding content.

3.1.5 Back-end

3.1.5.1 Background on OpenJML API

OpenJML is verification tool for Java programs to check the correctness of code with respect to specifications. OpenJML is built using the OpenJDK compiler, which is also an implementation of Java Modeling Language.

How does OpenJML work ?

The process that goes from the writing of a specification to getting it verified and sending of output to the console is described below;

STEP 1: The written preconditions become assumptions at the start of a procedure and postconditions become the assertions at the end.

STEP 2: The assertions, assumptions, and code are translated into a basic block form. This block form uses single-assignment in labeling of variables.

STEP 3: These basic blocks are further translated into verification conditions (VCs).

STEP 4: The VCs are expressed in *SMTLIBv2* format.

STEP 5: An SMT solver of choice is applied to all the VC's. *CVC4* is used primarily, and interoperability with *Z3* has also been demonstrated.

STEP 6: The logical variables of any counter-example that the SMT solver finds are converted back to source code variables and are shown in the source code editor with hover information and syntax highlighting.

SMT provers

Adapter code is necessary for interfacing with SMT solvers as they vary on the being compatible

with the SMT-LIB format. OpenJML has been tested with *SMT-LIBv2, Z3(v4.3), CVC4(v1.2)* and *Yices (v2.1)* on Windows and Mac.

Verification Types

Type-checking: The OpenJML tool works like a Java compiler. If we run OpenJML on a file with the argument '-no-purityCheck' as an argument, it will type-check JML and Java in the file and any classes that are dependent on the file.

Static Checking: Extended Static Checking (ESC) tries to prove the correctness of code statically (before running the program). This is done by using SMT solvers.

Runtime Assertion Checking: Runtime Assertion Checking translates JML assertions into dynamic checks during execution, all assertions are then checked while the program is being run; on finding a discrepancy, an error is thrown.

3.1.5.2 Docker Containers

Docker is an open-source project that makes use of much of the resource isolation features of Linux to create a sandbox for an application. These atomic units are called containers. Containers spin up an application with all of its dependencies in a single, portable, deployable unit. Multiple isolated containers can run on one host OS, without the need of hypervisor. This is useful in Rapid Application Development and Testing [12].

Components of Docker

The following are the components of Docker:

- Container – The application sandbox, which is based on an image of an OS and holds the necessary dependencies of an application. After launching a container, whenever changes

are made, a writable layer is added on top of the base image.

- Image – An Image is combination of file-system and other parameters that are used by a container to initialize. A static snapshot of this combination that is stored with the containers' recent configuration. The image is a read-only layer which is never modified, but can be added with a new layer. This will create a new image. A platform image, is an image with no parent. Platform images define the needed utilities, packages, and environment for the containers to run.

- Registry – Registries are repositories that contain all the images that are available for use.

- Dockerfile – A Dockerfile is a configuration file which has the build instructions to start a Docker container.

Why use Dockers?

The principal reason of using Dockers is owing to the portability and security that they provide. The various features [15] of portability and security that were considered are:

Rapid application deployment – Reduced size and minimal setup time makes application deployment easy.

Portability across machines – An application and all its dependencies bundled into a container, transferable to any system with the same host or with hypervisor access to the same host give Dockers an edge over legacy systems.

Version control and component reuse – Components can be reused from the previous layers of the containers, making it easy to roll-back to previous versions and much lightweight.

Sharing – Containers can be shared on the cloud solutions, through a private or a public repository.

3.1.5.3 NGINX

NGINX (*engine-x*) is and open-source, high-performance reverse proxy HTTP server and IMAP/POP3 proxy server. It is known for the features, stability, simplicity in configuration and low resource consumption with high performance. NGINX does not use threads to handle requests. Instead, it uses a much more scalable option of event-driven (async) architecture. It comes with a lightweight structure, which has a small memory footprint. Making NGINX scale from tiniest of virtual private server to huge server clusters.

Why use NGINX ?

In *TryOpenJML* application, NGINX was used when the application was running on *Amazon Web Services* and is still being used while it runs on the servers provided by the University of Central Florida.

3.2 Tutorial

There are some resources from where one can learn JML, the OpenJML User Guide, written by David Cok of GrammaTech Inc., is a work-in-progress guide to give an in-depth know-how of OpenJML, an older web-based version of the guide is available [16]. And, The JML Reference Manual [17] has been of great help in learning and building of this tutorial. The *TryOpenJML* tutorial, will be available on the web-application alongside with a text-based online JML code runner. It will consist of seven chapters/ lessons ranging from an Overview of JML to writing the first pre- and post-condition, to later introduction to working of JML annotations, Model and Ghost fields, Modifiers and Assignable Clauses and ending with Invariants.

CHAPTER 4: IMPLEMENTATION

4.1 Preview

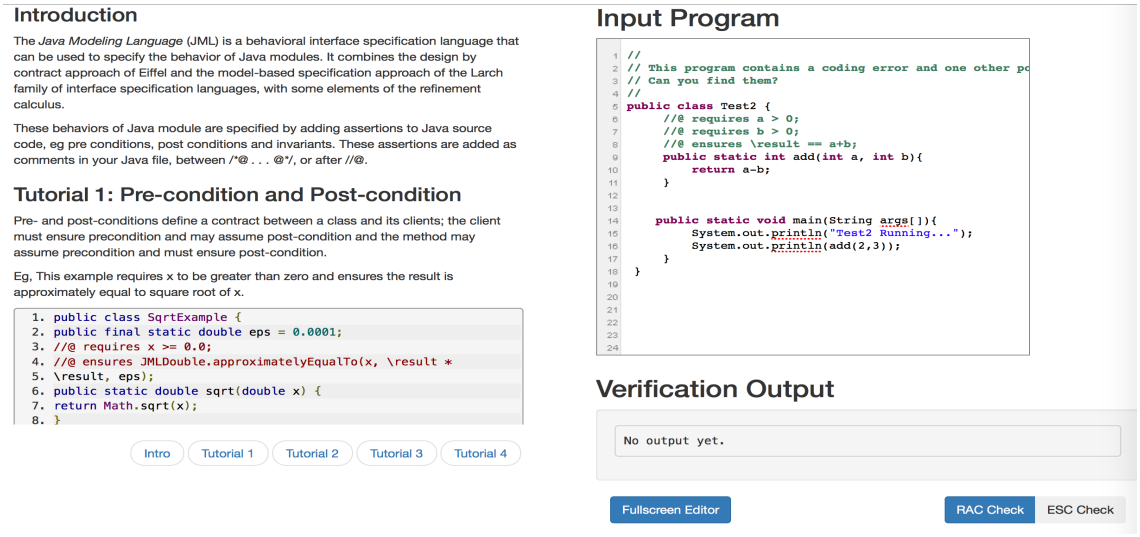

Figure 4.1: TryOpenJML web application preview

4.2 OpenJML Runner

TryOpenJML has an online code running tool, which allows the user to edit, compile and execute their Java/JML programs. The code running tool gives a user the place to get hands-on experience while they are initially learning JML. A full-screen editor has also been provided, for users who are well acquainted with OpenJML but want to compile and run their code.

Two types of checks are available in TryOpenJML, Runtime Assertion Check(RAC) and Extended Static Check(ESC). As these are the most important part of the JML runner, the working of these functions is explained below, with the help of sequence diagrams.

4.2.1 *Runtime Assertion Check (RAC)*

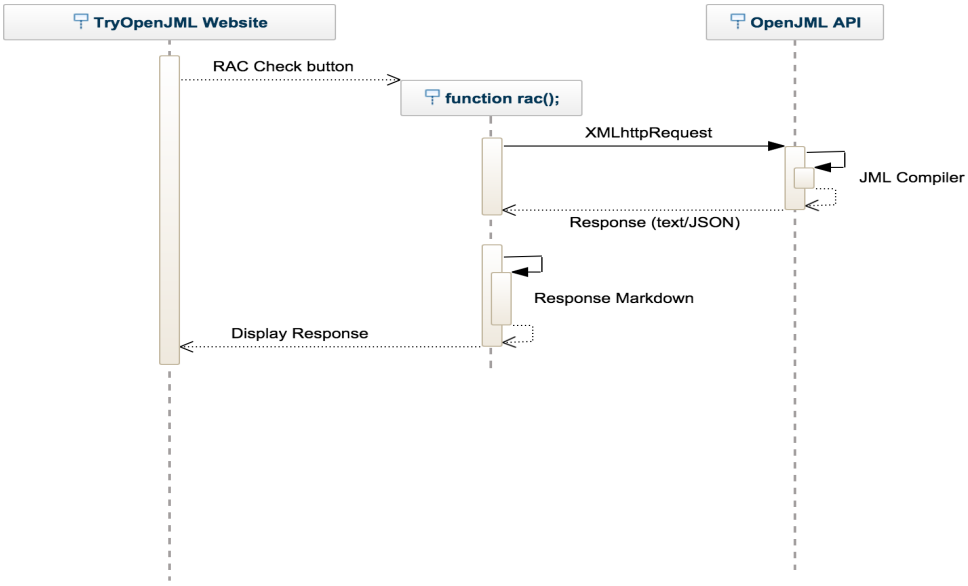

Figure 4.2: Runtime Assertion Check - Sequence Diagram

RAC Check button - This button is present on the TryOpenJML website for code runner with the tutorial as well as the full-screen editors. It triggers a JavaScript function *rac()*, on click.

The rac() function - This function handles HTTP requests and responses to and from OpenJML to run the OpenJML runtime assertion checker on the program. The API also takes care of converting responses to readable text. The process of converting the code in text format to JSON while sending the request and converting it back on a response is done by a lightweight markup language, Markdown.

OpenJML Compiler - The OpenJML Compiler verifies JML assertions on runtime and sends back the suitable message if the assertions are invalid.

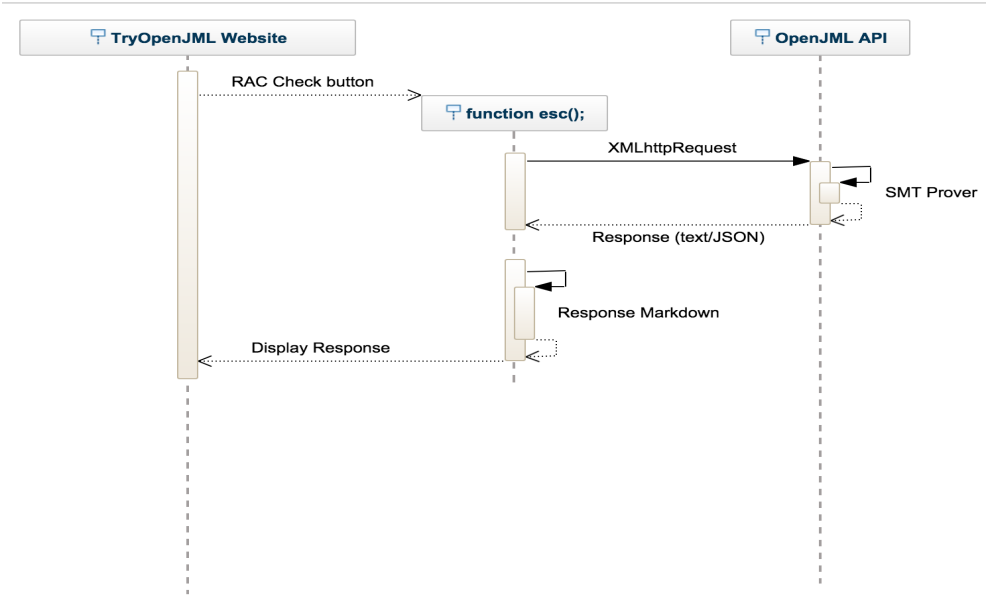

Figure 4.3: Extended Static Check - Sequence Diagram

4.2.2 Extended Static Check (ESC)

ESC Check button - This button is present on the TryOpenJML website for code runner with the tutorial as well as the full-screen editors. It triggers a JavaScript function *esc()*, on click.

The esc() function - This is a JavaScript function which handles HTTP requests and responses to and from the OpenJML API; it also takes care of converting them to readable text through Markdown.

SMT Provers - ESC uses SMT Provers like SMT-LIB, Z3 or CVC4 to statically checks the logic of the program and returns the result of the check to the *esc()* function.

4.3 Performance evaluation of the Verily framework

Loadster is used to run performance tests on the TryOpenJML application based on the Verily framework and the application which is based on Node.js framework. The configuration for testing the applications is shown in the figures 4.4 and 4.5.

Option	Value
Script	tojmlVerily
Engine	Built-in load engine
Virtual Users	25
Ramp Up	5 minutes, natural ramp up
Peak Duration	5 minutes
Ramp Down	5 minutes, natural ramp down
HTTP User-Agent	Loadster
Simulated Bandwidth	Full bandwidth
Resource Timeout (individual)	30000ms
Resource Timeout (total)	120000ms
Threads per Virtual User	3
Wait Times	As recorded

Figure 4.4: Test configuration for Verily based application under test [3]

Option	Value
Script	NonVerily
Engine	Built-in load engine
Virtual Users	25
Ramp Up	5 minutes, natural ramp up
Peak Duration	5 minutes
Ramp Down	5 minutes, natural ramp down
HTTP User-Agent	Loadster
Simulated Bandwidth	Full bandwidth
Resource Timeout (individual)	30000ms
Resource Timeout (total)	120000ms
Threads per Virtual User	3
Wait Times	As recorded

Figure 4.5: Test configuration for Node.js based application under test [3]

Parameters like errors, page hits, response times and web logs are taken into consideration while

comparing the two frameworks. The findings are discussed in detail in section 5.2.

4.4 Tutorials

The sub-sections consist of the verbatim of tutorials in the TryOpenJML application.

4.4.1 Overview of OpenJML

"The *Java Modeling Language (JML)* is a behavioral interface specification language" [17] which is used to specify the behavior of Java modules. Behavioral interface specification languages give code-level annotations such as pre-/post-conditions, invariants, and assertions to a program that allows the programmer to express various behaviors for program modules. A *module* is a Java class or interface. Java Modeling Language combines the design by contract approach of Eiffel and the model-based specification approach of the Larch family of interface specification languages, with some elements of the refinement calculus [17].

4.4.1.1 Assertion

Behaviors of a Java module are specified by adding assertions to Java source code; in particular, one can add preconditions and postconditions to methods and invariants to classes and interfaces. These assertions are added as individual JML comments in your Java file, between $/*@$ and $@*/$, or after $//@$ to the end of a line. Note that there cannot be any white-space between the start of the Java comment and the first at-sign (@), as $/* @$ or $// @$ is a Java comment that is not processed by JML.

4.4.1.2 *Using TryOpenJML*

Enter your code in the text box, then click run RAC for Runtime Assertion Checking or ESC for Extended Static Checking. For trying the examples given above, copy them into the code runner and click on RAC or ESC to run.

4.4.2 *Writing your first pre-condition*

Preconditions define assumptions a method may have about the state in which it is called; the client must establish the method's precondition; conversely, the method's code may assume its precondition. In the given program "*requires*" preceeds each precondition; the preconditions assume that variables PrincipalAmt, NumberOfYears, and RateOfIntr are all greater than zero.

```
1  //simpleIntr.java
2       public class simpleIntr {
3       //@ requires PrincipalAmt > 0;
4       //@ requires NumberOfYears > 0;
5       //@ requires 0 < RateOfIntr < 1;
6       //@ ensures \result = (PrincipalAmt * NumberOfYears * RateOfIntr)/100;
7       public static double Calc
8       (double PrincipalAmt, double NumberOfYears, double RateOfIntr)
9       {
10      return (PrincipalAmt * NumberOfYears * RateOfIntr)/100;
11      }
12      public static void main(String args[]){
13           System.out.println("Installment will be...");
14           System.out.println(Calc(1800000.00,3.50,12.75));
15      }
16   }
```

4.4.3 *Writing your first post-condition*

Postconditions define guarantees a method gives to its clients; the client may assume its postcondition; conversely, the method must guarantee its post-condition in all possible executions. As shown in the example, *ensure* verifies if the value returned by the method will be equal to the simple interest formula.

4.4.4 *How do JML annotations work?*

Each character passed as an argument to the function *changeCase(char c)* below is checked to satisfy the conditions given in the *requires* clause and the result which is returned should satisfy the *ensure* condition mentioned before the function. The keyword *also* signifies that a specification following the keyword is in addition to the specification prior to it.

```
1  //changeCaseExample.java
2  public class changeCaseExample {
3
4        //@ requires c >= 'A' && c <= 'Z';
5        //@ ensures \result >= 'a' && \result <= 'z';
6        //@ also
7        //@ requires c >= 'a' && c <= 'z';
8        //@ ensures \result >= 'A' && \result <= 'Z';
9        //@ also
10       //@ requires !( c >= 'A' && c <= 'Z') && !( c >= 'a' && c <= 'z');
11       //@ ensures \result == c;
12
13     public static char caseChange(char c) {
14         char result = ' ';
15         if( c > 'z'){
16           result = c;
17     } else if (c > 'z') {
18       result = (char)(c - 'a' + 'A');
19     } else if (c >= 'Z') {
20       result = 'c';
21     } else if(c >= 'A') {
```

```
22          result = (char)(c - 'A' + 'a');
23        } else {
24          result = c;
25        }
26        return result;
27        }
28        public static void main(String args[]){
29             System.out.println("Result is...");
30             System.out.println(caseChange('J'));
31             System.out.println(caseChange('M'));
32             System.out.println(caseChange('L'));
33
34         }
```

4.4.5 Model and Ghost fields

4.4.5.1 Model

The *model* modifier has two meanings. The first significance of a feature declared with the *model* is that it is present for specification purposes only. For example [17], a *model* field is only used for JML specifications and cannot be utilized in the Java code present outside the specifications. In a similar manner, a method with the *model* modifier is a method which is utilized only in annotations and not in Java code. The other significance of *model* depends on the type of feature that is being declared.

```
1 //@ public model float b;
2
3 //@ public model boolean empty;
4 protected int size = 0; //@ in empty;
5 //@ protected represents empty = (size == 0);
```

The 'model' modifier introduces a *specification-only* feature. For fields, it means that the field is only visible on the level of specification. In the above example, the *represents* clause defines, how the value of a model field is related to the implementation. The ghost and model modifiers are

mutually exclusive . Ghost fields will be covered in the later part of this section. There are some points to be noted about model fields, which are given below,

- A model field should not be declared to be final. If you think that a final model field is required in the program, then you should instead use a final ghost field instead.
- For an interface, a model field is static by default. Use 'instance' to declare a non-static field in an interface.

4.4.5.2 Ghost

Similar to a model field, a ghost field cannot be used in the Java code that is outside of JML specifications. Unlike a model field, the value of a ghost field is determined by its initialization or the value given to it in the most recent set-statement. The 'ghost' modifier introduces a *specification-only* field that is maintained by particular set statements.

```
/* By initialization*/
//@ ghost int i = 0;
//@ ghost int zero = 0, j, k = i+3;

/* By set-statement*/
//@ ghost boolean empty = true;
//@ set empty = (size == 0);
```

4.4.6 Modifiers and Assignable clauses

4.4.6.1 Modifiers

Modifiers in JML are similar to those in Java; it is just that they are only recognized as keywords in JML annotation comments [17].

4.4.6.2 *Assignable clauses*

In the example given below, the specification provided for the variable *array* will not be satisfied as a postcondition unless the function C is called. Now after function C is called, the postcondition would be satisfied. In this case, *assignable* clause is used.

Go ahead and use the code given and try to make it work.

Hint: psvm(String args[])

```
1  //A.java
2  public class A{
3        int array[];
4        //@ ensures array.length >=5;
5        //@ assignable array[];
6        public void B(){
7             array = new int[7];
8             C();
9        }
10       //@ ensures true;
11       public void C(){
12            array = new int[3];
13       }
14
15 }
```

4.4.7 *Invariants*

Invariants are the properties that must hold in all visible states. The understanding of visible states is important when dealing with invariants and constraints. " A state is *visible state* for an object *o*, if it occurs in one of the following stages of code's execution:" [17, section 8.2]

- " at the start of a non-helper non-static non-finalizer method invocation with *o* as the receiver." [17, section 8.2]

- "at the end of a non-helper non-static non-finalizer method invocation with *o* as the receiver," [17, section 8.2]
- "at the beginning or end of a non-helper static method invocation for a method in *o*'s class or some superclass of *o*'s class," [17, section 8.2] or
- "at the end of a non-helper constructor invocation that intializes *o*" [17, section 8.2]
- "when no constructor, destructor, non-static method invocation with *o* as the receiver, or static method invocation for a method in *o*'s class or some superclass of *o*'s class is in progress," [17, section 8.2]
- "at the beginning of a non-helper finalizer invocation, that is finalizing *o*." [17, section 8.2]

For example, the invariant in the example below has a *default (package)* visibility and in each state that is a visible state for an object of type *Invariant*, the object's field *bool* is not null and the array it refers to has exactly ten elements. No postcondition is required in this example, as the invariant is an implicit postcondition for the constructor.

```
1  //Invariant.java
2  package org.jmlspecs.samples.jmlrefman;
3  public abstract class Invariant {
4      boolean[] bool;
5      //@ invariant bool != null && bool.length == 10;
6
7      //@ assignable bool;
8      Invariant() {
9          bool = new boolean[10];
10     }
11 }
```

CHAPTER 5: FINDINGS

5.1 Web-based Application

Performance evaluation of the Verily Framework:

- A *socket timeout* error is thrown when the flow of continuous incoming data is stalled/broken. Nearly 4 percent of total errors in Verily based app is socket timeout error, and in Node.js based app it is 28 percent.
- A standard *HTTP 404* error is thrown when the client is connected to the server but is unable to find what is requested on the server. There were no *HTTP 404* errors found in the application created with the Verily framework, but nearly 70 percent of the total errors found in the Node.js framework were related to file not being found. This is because, the Verily framework takes care of any page/ file that is not found, from the beginning. A specific page can be displayed instead of Verily's default page by adding a Freemarker file named *404.ftl* in the *resources* folder of the Verily project.

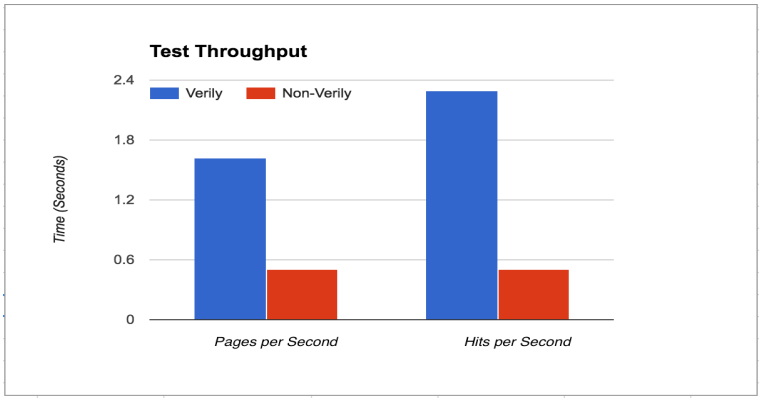

Figure 5.1: Test Throughput of Verily v/s Non-verily [3]

- The Verily based application gave more than three times the *hits per second* and was able to reach more than twice the *pages per second* than Node.js based application.
- The Verily framework scored better than the Javascript framework in the security test performed on the URLs. This test included, running scripts from the URL to reach the application by sending payloads through GET/ POST request. The Verily framework handles this by throwing HTTP 403 error. *HTTP 403 forbidden* error is thrown when the server understands the request, but refuses to authorize it.

Using the Verily Framework:

- Setting up the Verily framework was easy, the installation [18] is described in the form of steps with images.
- The Verily Framework is very straightforward and easily integrated with AngularJS with the help of its Javascript Bridge.
- The Method-Router-Response design needed some effort to understand but implementing the design was made effortless with the help of *goverily.org* [18].

Using AngularJS in the Verily framework:

- Setting up of single-page dynamic application was quicker than expected.
- Although learning the concepts of AngularJS took some time, development was fast once we were familiar with it.
- Two-way data binding was used to add features to the text-box, for the creation of the OpenJML code runner on the application. This aided in loading the runner with a sample on startup and edits were saved to the variable when the *RAC* or *ESC* check button was clicked.

Dockers were of great help in spinning up an environment for running the OpenJML API on AWS as well as on local servers. As dockers are lightweight, they were easy to migrate and would set up in seconds.

The following are the difficulties faced while creating the TryOpenJML app.

- The documentation to the Verily Framework is not yet complete which narrowed down the resources for reference. But with updates in the framework, I am sure there will be updates in documentation.
- As mentioned before, although basics of AngularJS were easy to learn and implement, but as we go deep into the concepts of directives, scopes, and two-way binding, the learning curve becomes very steep.
- Scopes in AngularJS were easy to use, but tricky to debug.
- In the case of Dockers, as it is a new and budding technology, there were fewer examples to refer. There will be lot more examples uploaded on web in the coming years, as the Docker community is spreading from just developers to QA to dev-ops.

5.2 Tutorials

With a hope that these Tutorials will be helpful to the users as there is an embedded code runner to practice JML as they move forward with the tutorials. More matter is going to be added to the tutorials in near future. It is discussed in detail, in Future Directions chapter of this thesis.

CHAPTER 6: FUTURE DIRECTIONS

There are going to be major updates on user experience and the content of the Tutorial in coming months. We have added a survey to gauge the impact of the tutorial application as well as what content should be added in future. Results of this study will aid in prioritizing revisions for TryOpenJML. Based on the tests performed, security aspect of the Verily Framework needs to be worked on in the future. Also interesting updates on the robustness of the Verily framework are expected in coming years. These revisions will have a great impact on the reliability aspect of TryOpenJML. We plan to verify these aspects with reliability metrics suitable for web applications.

CHAPTER 7: CONCLUSION

As we all know, web applications are a crucial part of the modern life. From managing our health-care, banking and social connections to almost everything is done through web applications either in the form of a mobile app or from a browser. Many web frameworks are available to us today, but almost none has been created taking reliability into consideration.

7.1 The Verily Framework

With the combination of application construction recipes and static analysis, Verily is designed to build more reliable web applications. The performance evaluation proved that, the Verily framework works well right out of the box as far as tackling of basic errors is concerned. Although Verily is currently under development, we can still use it successfully with emerging technologies and tools. All of Verily's recent development is found on the website *http://goverily.org*. There are more examples on Verily its Github page [19] and also a plugin has been released for IntelliJ 12.1.6 and higher, to ease the use of Verily in an IDE.

7.2 Tutorial on Java Modeling Language

The goal of the Tutorial is to convey the objective of Java Modeling Language through OpenJML, which is based on OpenJDK. It is a modeling language which goes hand in hand with the existing code. A wide range of tools are available for this, jmlunit for unit testing, jmlc for classes and jmldoc for web pages.

7.3 TryOpenJML - the web-based application for learning about JML

This research was conducted to accomplished to two goals. The first goal was to show how the Verily Framework could contribute to the domain of web application development by creating an application. And, the second goal was to create this application in a way that this app will aid in learning about Java Modeling Language. TryOpenJML is the web-based application that will help users to know more about JML. There are seven parts to the introductory tutorial, with many examples. There are a lot more chapters to be added to this tutorial, in the coming years.

APPENDIX : SOURCE CODE

Runtime Assertion Checker

Browser - Client-side Script

```
//Harness.js
 $scope.rac = function(){
    var tmp = $scope.racCheck;
    $scope.racCheck = "Checking... Please wait.";
    $scope.ajaxLoader = '/images/ajaxLoader.gif';
    $scope.output=
    $sce.trustAsHtml("<pre> loading Output . . .</pre>");
    $scope.$watch(function(scope){ return scope.program; },
    function()
    { console.log("digest called");  }, true);
    $http({
    url:'http://openjml.cs.ucf.edu/RuntimeAssertionChecker/run',
    data: {Source:$scope.program},
    method: 'POST',
    headers: {'Content-Type': 'application/json'}
    }).
    success(function(data, status, headers, config) {
    console.log(data);

    var response = data;
    $scope.racCheck = tmp;

    // find the markdown result
    var markdownContent = response.Outputs.filter(
    function(o){
    return o.MimeType==="text/x-web-markdown";})[0].Value;
    var str  = markdown.toHTML(markdownContent)
    .replace(/code>/g, "pre>");
    getIndicesOf("/tmp/", str, false);
    for(i=0;i<arrIndex.length;i++){
        str = str.slice(0, arrIndex[i])
        + str.slice(arrIndex[i]+15);
        for(j=i+1;j<arrIndex.length;j++){
            arrIndex[j]= arrIndex[j] - 15;
        }
    }
    var output = str;
    $scope.ajaxLoader= '/images/ajaxLater.gif';
    $scope.output= $sce.trustAsHtml("<pre>"+ output +"</pre>");
```

```
40
41      $scope.racCheck = "RAC Check";
42      }).
43      error(function(data, status, headers, config) {
44      console.log(data);
45      });
46     };
47 });
```

Method - RAC

```java
//RuntimeAssertionChecker.java
package methods;

import lang.InvalidClassfileException;
import runners.CheckerResult;
import runners.RACRunner;
import verily.lang.*;
import java.io.IOException;

public class RuntimeAssertionChecker{

     public static final CheckerResult run(String Source){

          RACRunner runner = new RACRunner(Source);

          try {
             return runner.run();
          } catch (IOException
          | InterruptedException
          | InvalidClassfileException e) {
             return new CheckerResult(e.getMessage(), false);
          }

     }

     public static final void metadata(){}
     public static final void language(){}
}
```

Router - RAC

```java
//RuntimeAssertionChecker.java
package routers;

import runners.CheckerResult;
import runners.RACRunner;
import verily.lang.*;
import java.util.HashMap;

public class RuntimeAssertionChecker{

     public static final Content
     run(String Source, CheckerResult result){

     Content response = new JSONContent();
     response
     .setContent(RACRunner
     .checkerResultToRise4RunFormat(result));
     return response;
     }

     public static final Content metadata(){
     Content response = new JSONContent();
     response
     .setContent(new TemplateHTMLContent("
     RuntimeAssertionCheckerMetadata.ftl", new HashMap())
     .getContent());
     return response;
     }

    public static final Content language(){
    Content response = new JSONContent();
    response
    .setContent(new TemplateHTMLContent("Language.ftl",
    new HashMap())
    .getContent());

    return response;
    }
}
```

Runner - RAC

```java
//RACRunner.java
package runners;

import org.json.simple.JSONArray;
import org.json.simple.JSONObject;
import org.json.simple.JSONValue;
import org.json.simple.parser.JSONParser;
import util.Constants;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.List;

/**
 * Created by jls on 5/13/2015.
 */
public class RACRunner extends Runner {


    public RACRunner(String source) {
        super(source);
    }

    public String[] getArgs(String fileName, String filePath){

        List<String> args = new ArrayList<String>();

        args.add("sh");
        args.add("-c");
        //args.add(String.format("cat %s | python
        //tools/tool_runner.py --timeout 20 -rac %s",
        //filePath.replaceAll("\\\\", "/"), fileName + //".java"));
//use the above code for executing code without Dockers

        args.add(String.format("cat %s | docker run -i
        openjml/try:v1 python /tools/tool_runner.py
        --docker --timeout 20 -rac %s",
        filePath.replaceAll("\\\\", "/"), fileName +
        ".java"));

        String[] ar = new String[args.size()];
```

```
44          return args.toArray(ar);
45
46      }
47
48      public static String
49      checkerResultToRise4RunFormat(CheckerResult result){
50
51          JSONObject o = (JSONObject) JSONValue
52          .parse(result.toJSON());
53          // rule 1
54          JSONObject response = new JSONObject();
55          response.put("Version", Constants.version);
56          JSONArray responses = new JSONArray();
57
58          if(result.getStatus()) {
59
60              response.put("stdout", o.get("stdout"));
61              response.put("returnCode", o.get("returnCode"));
62              response.put("fullOutput", o.get("fullOutput"));
63              response.put("timeout", o.get("timeout"));
64
65              // we coulnd't finish in time
66              if(((Boolean)o.get("timeout"))){
67                  JSONObject plain = new JSONObject();
68                  JSONObject md    = new JSONObject();
69
70                  plain.put("MimeType", "text/plain");
71                  plain.put("Value", "OpenJML could not verify
72                  your program before the timeout elapsed.
73                  Please try again (or write a smaller program)."
74                  + Constants.VERILY_TAGLINE_TXT);
75
76                  md.put("MimeType", "text/x-web-markdown");
77                  md.put("Value", "**OpenJML could not verify
78                  your program before the timeout elapsed.
79                  Please try again (or write a smaller program).**"
80                  + Constants.VERILY_TAGLINE_MD);
81
82                  responses.add(plain);
83                  responses.add(md);
84                  response.put("Outputs", responses);
85          }
86              // BEST case scenario. Return code is zero
87              //and there is no output
88
```

```
89              else if(((Long)o.get("returnCode"))==0
90              && ((String)o.get("stdout")).length()==0){
91      //|||((String)o.get("stdout")).contains("^")==false)
92                  JSONObject plain = new JSONObject();
93                  JSONObject md    = new JSONObject();
94
95                  plain.put("MimeType", "text/plain");
96                  plain.put("Value", "Your program appears
97                  to satisfy its specifications!"
98                  + Constants.VERILY_TAGLINE_TXT);
99
100                 md.put("MimeType", "text/x-web-markdown");
101                 md.put("Value", "**Your program appears to
102                 satisfy its specifications!**"
103                 + Constants.VERILY_TAGLINE_MD);
104
105                 responses.add(plain);
106                 responses.add(md);
107                 response.put("Outputs", responses);
108             }
109
110     // next case, return code is zero, but there is output
111             else {
112                 JSONObject plain = new JSONObject();
113                 JSONObject md    = new JSONObject();
114
115                 plain.put("MimeType", "text/plain");
116                 plain.put("Value", o.get("stdout")
117                 + Constants.VERILY_TAGLINE_TXT);
118
119                 md.put("MimeType", "text/x-web-markdown");
120                 md.put("Value", "```" + o.get("stdout")
121                 + "```" + Constants.VERILY_TAGLINE_MD);
122
123                 responses.add(plain);
124                 responses.add(md);
125                 response.put("Outputs", responses);
126             }
127
128         }else{
129
130             o.put("Version", Constants.version);
131             JSONObject plain = new JSONObject();
132             JSONObject md    = new JSONObject();
133
```

```
134                 plain.put("MimeType", "text/plain");
135                 plain.put("Value", "An error occurred while
136                 trying to verify your program (or there's a bug
137                 in OpenJML). Please send the program you were
138                 trying to verify to jls@cs.ucf.edu."
139                 + Constants.VERILY_TAGLINE_TXT);
140
141                 md.put("MimeType", "text/x-web-markdown");
142                 md.put("Value", "**An error occurred while
143                 trying to verify your program (or there's a bug
144                 in OpenJML). Please send the program you were
145                 trying to verify to jls@cs.ucf.edu.**"
146                 + Constants.VERILY_TAGLINE_MD);
147
148                 responses.add(plain);
149                 responses.add(md);
150                 o.put("Outputs", responses);
151                 return o.toJSONString();
152             }
153             return response.toJSONString();
154         }
155 }
```

Extended Static Checker

Browser - Client-side Script

```
//Harness.js
$scope.esc = function(){
var tmp = $scope.escCheck;
$scope.escCheck = "Checking... Please wait.";
$scope.ajaxLoader = '/images/ajaxLoader.gif';
$scope.output= $sce.trustAsHtml("<pre> loading Output . .
.</pre>");
$scope.$watch(function(scope){
return scope.program; },
function(){ console.log("digest called");  }, true);

$http({
    url:'http://openjml.cs.ucf.edu/ExtendedStaticChecker/run',
    data: {Source:$scope.program},
    method: 'POST',
    headers: {'Content-Type': 'application/json'}
}).
    success(function(data, status, headers, config) {
    console.log(data);

    var response = data;
    $scope.escCheck = tmp;
    // find the markdown result
    var markdownContent = response.Outputs
    .filter(function(o)
    { return o.MimeType==="text/x-web-markdown";})[0].Value;
    var str  = markdown.toHTML(markdownContent)
    .replace(/code>/g, "pre>");
    getIndicesOf("/tmp/", str, false);
    for(i=0;i<arrIndex.length;i++){
        str = str.slice(0, arrIndex[i])
        + str.slice(arrIndex[i]+15);
        for(j=i+1;j<arrIndex.length;j++){
            arrIndex[j]= arrIndex[j] - 15;
        }
    }
    output = str;
    $scope.ajaxLoader= '/images/ajaxLater.gif';
    $scope.output= $sce.trustAsHtml("<pre>"+ output +"</pre>");
```

```
40      $scope.escCheck = "ESC Check";
41      }).
42      error(function(data, status, headers, config) {
43      console.log(data);
44      });
45  };
```

Method - ESC

```java
//ExtendedStaticChecker.java
package methods;

import lang.InvalidClassfileException;
import runners.CheckerResult;
import runners.ESCRunner;
import verily.lang.*;
import java.io.IOException;

public class ExtendedStaticChecker{

    public static final CheckerResult run(String Source){

         ESCRunner runner = new ESCRunner(Source);
         try {

             return runner.run();
         } catch (IOException
         | InterruptedException
         | InvalidClassfileException e) {

        return new CheckerResult(e.getMessage(), false);
         }

     }

    public static final void metadata(){}
    public static final void language(){}

}
```

Router - ESC

```
//ExtendedStaticChecker.java
package routers;

import runners.CheckerResult;
import runners.ESCRunner;
import verily.lang.*;
import java.util.HashMap;

public class ExtendedStaticChecker{

     public static final Content run(String Source,
     CheckerResult result){
     Content response = new JSONContent();
     response.setContent(ESCRunne
     .checkerResultToRise4RunFormat(result));
     return response;
     }

    public static final Content metadata(){
    Content response = new JSONContent();
    response.setContent(new TemplateHTMLContent("
    ExtendedStaticCheckerMetadata.ftl",
    new HashMap()).getContent());
    return response;
    }

    public static final Content language(){
    Content response = new JSONContent();
    response.setContent(new TemplateHTMLContent
    ("Language.ftl",new HashMap()).getContent());
    return response;
    }
}
```

Runner - ESC

```java
//ESCRunner.java
package runners;

import org.json.simple.JSONArray;
import org.json.simple.JSONObject;
import org.json.simple.JSONValue;
import org.json.simple.parser.JSONParser;
import util.Constants;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.List;

public class ESCRunner extends Runner {

    public ESCRunner(String source) {
        super(source);
    }

    public String[] getArgs(String fileName, String filePath){

        List<String> args = new ArrayList<String>();
        args.add("sh");
        args.add("-c");

        //
        // use this to toggle between local and docker
        //mode
        //
        //args.add(String.format("cat %s | python
        //tools/tool_runner.py --timeout 20 -esc %s",
        //filePath.replaceAll("\\\\", "/"), fileName +
        //".java"));

        args.add(String.format("cat %s | docker run -i
        openjml/try:v1 python /tools/tool_runner.py
        --docker --timeout 20 -esc %s",
        filePath.replaceAll("\\\\", "/"), fileName +
        ".java"));

        String[] ar = new String[args.size()];
```

```
44          return args.toArray(ar);
45
46      }
47
48      public static String checkerResultToRise4RunFormat(
49      CheckerResult result){
50
51          JSONObject o = (JSONObject)
52          JSONValue.parse(result.toJSON());
53
54          // rule 1
55          JSONObject response = new JSONObject();
56          response.put("Version", Constants.version);
57          JSONArray responses = new JSONArray();
58
59          if(result.getStatus()) {
60
61              response.put("stdout", o.get("stdout"));
62              response.put("returnCode", o.get("returnCode"));
63              response.put("fullOutput", o.get("fullOutput"));
64              response.put("timeout", o.get("timeout"));
65
66              // we coulnd't finish in time
67              if(((Boolean)o.get("timeout"))){
68                  JSONObject plain = new JSONObject();
69                  JSONObject md    = new JSONObject();
70
71                  plain.put("MimeType", "text/plain");
72                  plain.put("Value", "OpenJML could not
73                  verify your program before the timeout
74                  elapsed. Please try again (or write a
75                  smaller program)." +
76                  Constants.VERILY_TAGLINE_TXT);
77
78                  md.put("MimeType", "text/x-web-markdown");
79                  md.put("Value", "**OpenJML could not verify
80                  your program before the timeout elapsed.
81                  Please try again (or write a smaller
82                  program).**" +
83                  Constants.VERILY_TAGLINE_MD);
84
85                  responses.add(plain);
86                  responses.add(md);
87                  response.put("Outputs", responses);
88
```

```
89              }
90              // BEST case scenario. Return code is zero and
91              //there is no output
92
93              else if(((Long)o.get("returnCode"))==0
94              && ((String)o.get("stdout")).length()==0){
95
96                  JSONObject plain = new JSONObject();
97                  JSONObject md    = new JSONObject();
98                  plain.put("MimeType", "text/plain");
99                  plain.put("Value", "Your program appears to
100                 satisfy its specifications!" +
101                 Constants.VERILY_TAGLINE_TXT);
102
103                 md.put("MimeType", "text/x-web-markdown");
104                 md.put("Value", "**Your program appears to
105                 satisfy its specifications!**" +
106                 Constants.VERILY_TAGLINE_MD);
107
108                 responses.add(plain);
109                 responses.add(md);
110                 response.put("Outputs", responses);
111             }
112
113             // next case, return code is zero, but there is
114             //output
115             else {
116                 JSONObject plain = new JSONObject();
117                 JSONObject md    = new JSONObject();
118                 plain.put("MimeType", "text/plain");
119                 plain.put("Value", o.get("stdout") +
120                 Constants.VERILY_TAGLINE_TXT);
121
122                 md.put("MimeType", "text/x-web-markdown");
123                 md.put("Value", "```" + o.get("stdout") +
124                 "```" + Constants.VERILY_TAGLINE_MD);
125
126                 responses.add(plain);
127                 responses.add(md);
128                 response.put("Outputs", responses);
129             }
130
131         }else{
132
133             o.put("Version", Constants.version);
```

```
134                 JSONObject plain = new JSONObject();
135                 JSONObject md    = new JSONObject();
136
137                 plain.put("MimeType", "text/plain");
138                 plain.put("Value", "An error occurred while
139                 trying to verify your program (or there's a bug
140                 in OpenJML). Please send the program you were
141                 trying to verify to jls@cs.ucf.edu." +
142                 Constants.VERILY_TAGLINE_TXT);
143
144                 md.put("MimeType", "text/x-web-markdown");
145                 md.put("Value", "**An error occurred while
146                 trying to verify your program (or there's a bug
147                 in OpenJML). Please send the program you were
148                 trying to verify to jls@cs.ucf.edu.**" +
149                 Constants.VERILY_TAGLINE_MD);
150
151                 responses.add(plain);
152                 responses.add(md);
153                 o.put("Outputs", responses);
154                 return o.toJSONString();
155             }
156             return response.toJSONString();
157         }
158 }
```

LIST OF REFERENCES

[1] TryOpenJML Github page. https://github.com/openjml/try-openjml/. 2014.

[2] Freemarker Template Language(FTL) Website. http://freemarker.org/. 2016.

[3] Inc. Brickyard Technologies. https://www.loadsterperformance.com/. 2016.

[4] John L. Singleton and Gary T. Leavens. Verily: A web framework for creating more reasonable web applications. pages 560–563, 2014.

[5] Gary T. Leavens. Tutorial on jml, the java modeling language. 2007.

[6] Gary T. Leavens, Joseph R. Kiniry, and Erik Poll. Computer aided verification: 19th international conference, cav 2007, berlin, germany, july 3-7, 2007. proceedings. pages 37–37, 2007.

[7] David R. Cok. Nasa formal methods: Third international symposium, nfm 2011, ca, usa, april 18-20, 2011. proceedings. pages 472–479, 2011.

[8] Jeff Tian and Li Ma. Web testing for reliability improvement. 67:177 – 224, 2006.

[9] Li Ma and Jeff Tian. Web engineering: International conference, icwe 2003 oviedo, spain, july 14–18, 2003 proceedings. pages 314–323, 2003.

[10] Wenbin Jiang, Hao Dong, Hai Jin, Hui Xu, and Xiaofei Liao. Pervasive computing and the networked world: Joint international conference, icpca/sws 2012, istanbul, turkey, november 28-30, 2012, revised selected papers. pages 282–296, 2013.

[11] Ivan Melia; Sandeep Puri; Ken Owens; Kiran Thirumalai, Sailesh Yellumahanti from Cisco, Lars Herrmann; Mark Coggin; Joe Fernandes; Kimberly Craven, and Dan Juengst from Red Hat. Linux containers: Why theyre in your future and what has to happen first. 2014.

[12] Brandon Chavis and Thomas Jones. Docker on aws - running containers in the cloud. 2015.

[13] AngularJS Website. https://docs.angularjs.org/guide/introduction. 2016.

[14] Bootstrap CSS Website. http://getbootstrap.com/css/. 2016.

[15] RedHat Enterprise Linux Website. https://access.redhat.com/documentation/en-us/. 2016.

[16] OpenJML Website. http://www.openjml.org/. 2015.

[17] JML Reference Manual. http://www.eecs.ucf.edu/ leavens/jml/jmlrefman/jmlrefmantoc.html. 2014.

[18] Th Verily Framework Documentation. http://docs.goverily.org/en/latest/quickstart.html. 2014.

[19] John Singleton's Github page. https://github.com/jsinglet/verilybidding. 2014.