Pex Extension for Generating User Input Validation Code for Web Applications

Karel Frajták, Miroslav Bureš and Ivan Jelínek
Department of Computer Science, Faculty of Electrical Engineering, Czech Technical University, Karlovo nám. 13, 121 35 Praha 2, Czech Republic

Keywords: User Input Validation, Code Generation, Web Application Testing.

Abstract: The code written by a software developer is not always flawless. The more code is created the more errors are introduced into the system. In web development different programming languages can be used to implement back–end and front–end sides of the application. For example, it is possible to implement user input validation multiple times — it validates the input values on client–side using JavaScript before the data is sent to server and then the received data is validated again on the server–side. The logic is duplicated, changes made to the validation code must be synchronised on both sides. All implementations must be also unit tested, which increases the time required to create and maintain multiple sets of unit tests. In this paper, we will describe how white–box testing tool Pex can be extended to generate user input validation code for ASP.NET MVC web applications. The validation code won't be duplicated in JavaScript on the client–side and the application will be protected from sending invalid input values from the client–side. The testers can focus on testing using meaningful data input values. Testing of corner cases can be automated thus saving the available resources — testers involved in testing and time spent on testing.

1 INTRODUCTION

Web applications are usually hybrid applications — more than one programming language is used in the development process. Code of back–end (server–side) part of the application can be written in high level programming language, like Java, C#, Ruby, Python, or PHP. Data–driven web applications where data is stored in database require the knowledge of database programming language. The client facing part of the hybrid web application, the front–end, is created using HTML with the help of JavaScript to improve the user experience. Other platforms suitable to create the front–end are available too, like Flash or Silverlight, but HTML and JavaScript combination is the most widespread. Knowledge of multiple programming languages is therefore essential.

Of course there are exceptions when one single programming language can be used to implement both sides of the web application. WebSharper (Bjornson et al., 2011) web framework builds both client and server–side of the web applications with functional programming language F#. All the code is written in F# only. Server–side is powered by F# code, special constructs are used to translate F# code into JavaScript code that is executed on the client–side. When developing application for Silverlight application platform single .NET Framework language is required. An alternative in Java world is Google Web Toolkit (GWT) (Tacy et al., 2013) where parts of the code written in Java are compiled to JavaScript to generate the front–end code.

In hybrid web applications the validation code is implemented on client–side to improve user experience. The validation code must be also implemented on the server–side too — the data received from the client side can not be trusted and must be validated. Validation code is duplicated on both sides.

In this paper we propose an improvement to web application unit testing. The client–side JavaScript code duplicates the logic already created on server–side. This code must be also properly tested — developer has to create and maintain another set of unit tests. Our proposed solution uses white–box testing tool to generate the client–side validation code from the server–side validation code. The tool generates the unit tests for server–side code, extracts the validation logic and translates it to JavaScript code. This code then validates the user input values on client–side.

Frajták K., Bureš M. and Jelínek I..
Pex Extension for Generating User Input Validation Code for Web Applications.
DOI: 10.5220/0004994103150320
In *Proceedings of the 9th International Conference on Software Engineering and Applications* (ICSOFT-EA-2014), pages 315-320
ISBN: 978-989-758-036-9

We're going to discuss the problems of creating unit tests to achieve high test coverage in the next section and briefly describe the white–box testing method. Our motivation and the reasons we have decided to use Pex and create our extension is explained in section 4. Our proposed solution is described in next section 5. And section 6 concludes the results of our work and describes our future work.

2 IN PURSUIT OF HIGH TEST COVERAGE

Code that is unit tested since the very beginning is not error–free, but the number of errors is reduced. Unit tests are in most cases created by the developers at the moment of writing the code or in worse case much later. They are familiar with their code and know the details of it and know how to write the tests quickly. The pitfall is that they can introduce a false positive error. The test does fail to indicate a problem or failure but it does not verify the correctness of the application code. They might have not fully understood the requirements or the test was created to test their own incorrect code and system specifications and requirements were ignored. They have created a non–failing test that verifies incorrect behaviour.

Code coverage is a metric used to describe the degree to which the source code of a program is tested by a test suite. The higher the number, the better. But it is quite difficult to create unit tests manually with very high test coverage. The most difficult is testing the code paths that result in an exception being explicitly thrown — when constraint is broken or a contract validation fails. Tester must set up the test environment properly and infer the set of input variables values that when passed to the method will throw an exception. That means he has to have access to the source code and be familiar with it. Significant amount of time can be spent on analysing the code and writing the test case than actual testing.

These corner cases can be tested using white–box testing tool. We have decided to use a white–box testing tool in our proposed system as it gives a better insight into the internal structure of the tested code and it can infer the minimal sets of significant input parameter values.

2.1 Insight Into the Code

White–box tool test internal structures of an application, in contrast to black–box testing when only the public interfaces of the application are tested. The internals of an application can be analysed statically or dynamically. Both having its advantages and disadvantages but supplementing each other. Static analysis analyses the code at a higher level than dynamic analysis and can detect issues that are correct from the viewpoint of dynamic analysis(Karpov, 2011).

White–box test case generators produce representative set of tests that can cover most of the application code. Test case generator with insight into the code minimizes the number of generated test cases to achieve high test coverage.

JWalk(Simons, 2007) is white–box testing tool targeting Java platform. It supports lazy systematic unit testing. The tool infers dynamically the evolving specification of an application on the fly by dynamic analysis and uses the class state exploration and then it tests the code systematically and exhaustively to bounded depths.

Pex(Tillmann and de Halleux, 2008) generates inputs for parametrized unit tests by analysing the branch conditions in the .NET platform program. Test inputs are chosen based on whether they can trigger new branching behaviours of the program. The analysis using a dynamic symbol execution (DSE) (Cadar and Sen, 2013) is an incremental process. It refines a predicate along the execution path over the formal test input parameters over the set of behaviours observed previously. Variants of the previous inputs are inferred by a constraint solver Z3 (de Moura and Bjørner, 2008) in order to steer future program executions along alternative program path.

White–box testing tools help the developers to create unit tests to test the code on low level. However it is important for the application to be tested on higher levels — to meet the specification requirements, passing user acceptance testing, etc.

3 RELATED WORK

The solution proposed in (Alkhalaf et al., 2012) uses minimum and maximum input validation policies to validate user input. These policies are defined as regular expressions the input value must match. The JavaScript validation must be defined beforehand and the extraction is done in runtime, the extracted data is then analysed using string analysis. In (server side, 2012) the authors are parsing the HTML code of the page to locate the form and the input elements. For every element a metadata is created and stored.

As observed in (Jamrozik et al., 2013), Pex does not produce tests sensitive enough to make a good regression test suite, sometimes the end users expect different value or more values fitting given the condition or program path. A new tool extending Pex was

proposed with the use of augmented dynamic symbol execution (ADSE)(Jamrozik et al., 2012). ADSE is based on exploration of a program through path conditions, but before test data is generated, the conditions are augmented with additional conditions; multiple test data for the same path condition are derived.

We propose to use existing code without duplication to create user interface validation code to verify user input before triggering an action or computation. The JavaScript code will be generated automatically from the server–side code. It is easier to analyse server–side C# code than parsing HTML pages and locating input elements.

4 MOTIVATION

On an e–commerce web site user would like to complete his order. He is asked for his contact details, shipping address and credit card details in checkout process. He has to fill several web forms. User does not want to fill all the data first, submit the form, and wait for the error message when something was wrong with his data. To improve the usability it is desired to display an human–readable message to the user immediately when some of the provided values are invalid and must be corrected. The code on the client–side validates the values entered by the user preventing the form to be submitted when data is not valid.

We are going to demonstrate Pex capabilities on a sample method in Listing 2. First the method validates user's registration data, then it creates and saves user and returns the view when everything is all right. An exception is thrown when validation of input values fails.

```
return userName != (string)null &&
  7 < userName.Length &&
  password != (string)null &&
  passwordConfirmation != (string)null &&
  (password == passwordConfirmation ||
    password.Length ==
      passwordConfirmation.Length &&
        password.op_Equality((object)
          passwordConfirmation) != (sbyte)0) &&
  17 < 2014 - dateOfBirth &&
  2014 - dateOfBirth < 121;
}
```

Listing 1: Example of validation code.

Pex generates unit tests for the trivial observed values (see Figure 5) for input parameters. If all conditions are met then the path condition string at the point of returning boolean `true` value can be observed (see Listing 1).

Looking at the path condition for year of birth we can see the power of Pex observations. The required age for registered user ranges from 18 as the minimum value and 121 as the maximum value. Pex does not generate tests for all values from this range, but observes the condition and infers the proper code for the value to match this constraint. These basic Pex outputs can not be used to generate user input validation code.

5 PROPOSED SOLUTION

Our solution uses combination of techniques to achieve our goal. The process is described as follows:

1. Methods handling incoming user requests are identified – the method must be declared in a controller class and it must handle POST requests.
2. Unit tests similar to one in Listing 3 are created for each method — the header of the unit test method is basically header of the inspected method and all the required information to generate the code can be fetched by reflection API. We have added a special line to generated test method that forces Pex to output observed path conditions (see the last line of code in Listing 3). The observed path conditions are otherwise hidden.
3. Pex explores the method code and our extension analyses the code of the generated test methods and outputs the method headers and the observed path conditions into temporary storage (an XML document). The method header uniquely identifies the location in code and also the form on the page — every form is identified by an action URL which identifies the controller and action handles this form data.
4. Extracted C# code is passed to C#/JavaScript compiler.
5. Generated JavaScript code is embedded into the assembly to be used later.

All the values of input element are passed to the validation function (see the code of `submitHandler` method in Listing 4) on client–side. If the validation condition is met, the form is submitted. The form and the validation logic were generated for the method shown in Listing 2. All form input values are validated on form submit.

```
public ActionResult Register(string userName, string password, string passwordConfirmation, int yearOfBirth)
{
  if (string.IsNullOrEmpty(userName)) throw new ArgumentNullException("userName");
  if (userName.Length < 8) throw new ArgumentException("User must be at least 8 characters long");
  if (password == null) throw new ArgumentNullException("password");
  if (passwordConfirmation == null) throw new ArgumentNullException("passwordConfirmation");
  if(password != passwordConfirmation) throw new ApplicationException("Password does not match the confirm
       password.");
  var thisYear = DateTime.Today.Year;
  if (thisYear - yearOfBirth < 18 || thisYear - yearOfBirth > 120)
    throw new ApplicationException("Invalid year of birth");
  new UserService().CreateUser(userName, password);
  return View();
}
```

Listing 2: Sample method.

```
[TestClass, PexClass(typeof(UserController))]
public partial class UserControllerTest
{
  [PexMethod]
  public void Register(
    string userName, string password, string
        passwordConfirmation, int yearOfBirth)
  {
    var uc = new UserController();
    PexAssert.IsTrue(
      uc.Register(userName, password,
          passwordConfirmation, yearOfBirth));
    PexObserve.ValueForViewing(
      "path condition", PexSymbolicValue.
          GetPathConditionString());
  }
}
```

Listing 3: Test method used for Pex explorations.

```
function __validate(userName, password,
     passwordConfirmation, dateOfBirth)
{
  return
    userName !== null && 7 < userName.length &&
        password !== null && passwordConfirmation
        !== null && (password ===
        passwordConfirmation || password.length ===
         passwordConfirmation.length && password
        === passwordConfirmation) &&
       17 < 2014 - dateOfBirth &&
       2014 - dateOfBirth < 121;
  }
}

$('#form').validate({
  submitHandler: function(form) {
    var _1 = $("#userName").val();
    var _2 = $("#password").val();
    var _3 = $("#passwordConfirmation").val();
    var _4 = $("#yearOfBirth").val();
    if(!__validate(_1, _2, _3, _4))
    {
      // display error message
      return false;
    }
  }
});
```

Listing 4: Generated validation code.

We have hit the limit of the current version when the compilation of the extracted code to JavaScript either failed (given constructs were not available) or the generated JavaScript code was not valid and errors were reported on the client–side. This happened when the validation code for user input validation methods used code that can not be evaluated on client–side, e.g. code accessing database, detecting the existence of a file or evaluating session values. We had to skip the code generation for these methods.

5.1 Validation

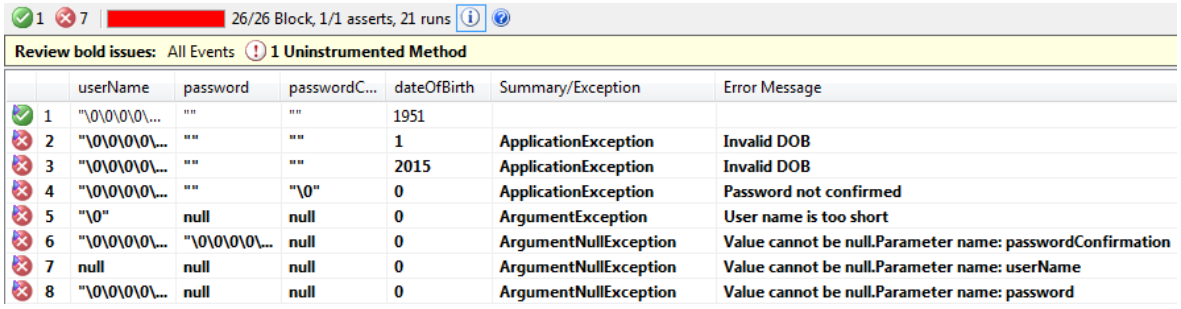
The correctness of the auto–generated JavaScript code is critical for our solution to work seamlessly. We have used Karma[1], a JavaScript test–runner, to verify the correctness of the code:

1. a unit test case is generated for every JavaScript validation method generated by our extension
2. the values discovered by Pex are supplied to this test case
3. the result is verified

This set of tests is executed only when a change is made to the application code and all the validation methods are re–generated. We have experienced some difficulties when using the values Pex discovered as an input to JavaScript test case — particularly speaking about date and time values and different representations of these data types in C# and JavaScript.

It might look like we are adding new tests and increasing the number of executed tests, but all these tests are generated and executed automatically, no tester is involved in this process. These tests validate our approach.

[1] http://karma-runner.github.io/0.8/index.html



1 7 26/26 Block, 1/1 asserts, 21 runs

Review bold issues: All Events 1 Uninstrumented Method

		userName	password	passwordC...	dateOfBirth	Summary/Exception	Error Message
✓	1	"\0\0\0\0\...	""	""	1951		
✗	2	"\0\0\0\0\...	""	""	1	ApplicationException	Invalid DOB
✗	3	"\0\0\0\0\...	""	""	2015	ApplicationException	Invalid DOB
✗	4	"\0\0\0\0\...	""	"\0"	0	ApplicationException	Password not confirmed
✗	5	"\0"	null	null	0	ArgumentException	User name is too short
✗	6	"\0\0\0\0\...	"\0\0\0\0\...	null	0	ArgumentNullException	Value cannot be null.Parameter name: passwordConfirmation
✗	7	null	null	null	0	ArgumentNullException	Value cannot be null.Parameter name: userName
✗	8	"\0\0\0\0\...	null	null	0	ArgumentNullException	Value cannot be null.Parameter name: password

Figure 5: Pex explorations output for method in Listing 2.

5.2 Experimental Verification

We have tested our tool on selected applications — the citations management application and selected parts of an e–commerce project TheBeerHouse[2]. The first application was developed using test driven development process, the TheBeerHouse project source code is shipped with number of unit tests.

First we used Pex in the way it was designed — to automatically generate test suites with high code coverage for the server–side code. Pex can also generate tests testing corner case scenarios (see the lines with red icon in Figure 5) and there was no need to create these tests manually. After that our extension was used to generate the test method used to extract the validation code and to convert it to JavaScript.

The generated code was not fully optimal and in some case we had to modify the code of the application for the JavaScript code to be generated without any problems and sometimes we had to processing the method. We have observer that the whole process is not very smooth and straightforward, but overall the results are promising.

The JavaScript code is hard to read and understand and it might be difficult to refactor and reuse the parts of the code later. This code was generated from a code of the tested application so we the readbility is not the most important aspect of this code. The code is also often regenerated.

5.3 Group Experiment

We have asked a team of 20 developers working on in–house developed project to participate in our experiment — group *A* of 10 developers was using our extension, while group *B* was not. After setting up the environment and brief introduction they started to use our Pex extension. We also ask them to report any issues. To avoid any misinterpretation of the results they have been working on the same tasks using the same code (separate branches were created for each group). In the end of trial period we asked them to answer survey questions and evaluate the experience.

- 80% of the selected developers think that the extension is useful and helpful
- 60% think the extension contributed to their work and simplified the testing process
- they also welcome the elimination of writing the same code twice
- 90% think that the process is not very straightforward and should be improved
- compared to the group *B* they were able to complete more tasks in shorter time and finish the work on time despite of the later start

6 CONCLUSIONS AND FUTURE WORK

In this paper we have described our Pex extension that improves the web application unit testing process. The extension leverages the code insight of a white–box testing tool to extract the code validating user input values on server–side and generates client–side validation code. It is no longer needed to duplicate the validation code on both sides and mirror the changes made to the code. The extracted code is then processed and translated to Javascript methods. The developer can focus more on server–side code only without writing the same code (in Javascript) for client–side. Our solution improves his development process and helps him to finish his tasks sooner.

Our solution has known limitations — we do not support code generation for methods with constraints that can not be evaluated on client–side without additional server request, e.g. code accessing file system or reading server session variables. Another limitation is that we can not validate more than one form on a web page. Our tool also fails to generate code for

[2]http://thebeerhouse.codeplex.com/

methods using C# constructs that can not be translated to JavaScript. In the future we would like to overcome these problems and make the whole process transparent by integrating the code extraction and generation into build process.

ACKNOWLEDGEMENTS

This research has been supported by MŠMT under research program No. 6840770014 and by Grant Agency of the CTU in Prague under grant SGS14/076/OHK3/1T/13.

REFERENCES

Alkhalaf, M., Bultan, T., and Gallegos, J. L. (2012). Verifying client-side input validation functions using string analysis. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 947–957, Piscataway, NJ, USA. IEEE Press.

Bjornson, J., Tayanovskyy, A., and Granicz, A. (2011). Composing reactive GUIs in F# using WebSharper. In *Implementation and Application of Functional Languages*, volume 6647 of *Lecture Notes in Computer Science*, pages 203–216. Springer Berlin Heidelberg.

Cadar, C. and Sen, K. (2013). Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90.

de Moura, L. and Bjørner, N. (2008). *Z3: An Efficient SMT Solver Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963/2008 of *Lecture Notes in Computer Science*, chapter 24, pages 337–340. Springer Berlin, Berlin, Heidelberg.

Jamrozik, K., Fraser, G., Tillman, N., and Halleux, J. (2013). Generating test suites with augmented dynamic symbolic execution. In Veanes, M. and Viganò, L., editors, *Tests and Proofs*, volume 7942 of *Lecture Notes in Computer Science*, pages 152–167. Springer Berlin Heidelberg.

Jamrozik, K., Fraser, G., Tillmann, N., and Halleux, J. D. (2012). Augmented dynamic symbolic execution. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 254–257, New York, NY, USA. ACM.

Karpov, A. (2011). Myths about static analysis. the third myth - dynamic analysis is better than static analysis @ONLINE. http://www.viva64.com/en/b/0117/. Accessed: 2013-09-04.

server side, a. (2012). Automated server-side form validation. In *Informatics, Electronics Vision (ICIEV), 2012 International Conference on*, pages 61–64.

Simons, A. (2007). JWalk: a tool for lazy, systematic testing of java classes by design introspection and user interaction. *Automated Software Engineering*, 14(4):369–418.

Tacy, A., Hanson, R., Essington, J., and Tokke, A. (2013). *GWT in Action*. Manning Publications.

Tillmann, N. and de Halleux, J. (2008). Pex-white box test generation for .NET. In *TAP*, pages 134–153.