Position Papers of the Federated Conference on
Computer Science and Information Systems pp. 117–122
DOI: 10.15439/2015F392
ACSIS, Vol. 6

# Using the Interaction Flow Modelling Language for Generation of Automated Front–End Tests

Karel Frajták, Miroslav Bureš, Ivan Jelínek
Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University
Karlovo nám. 13, 121 35 Praha 2, Czech Republic
Email: {frajtak, buresm3, jelinek}@fel.cvut.cz

***Abstract*—In the paper we explore the possibilities of automated test-case generation from the IFML model of application front–end. As opposed to the previous core UML standard, IFML captures the structure and properties of the application user interface, which gives us new possibilities in model–based test case generation: produced test cases have a higher probability of being consistent and of respecting the real feasibility of the tests in the tested application. In the presented solution we leverage the capabilities of an IFML model to capture details of front–end components to generate front–end automated tests, exercising particular actions in the tested application front–end to verify its expected behaviour according to an IFML model. The approach is based on the transformation of an IFML model to an application front–end test model — a more straightforward structure for the automated generation of test cases. Then, based on the defined rules, the abstract test cases are created from the model. The abstract test cases are then transformed using a template engine, to particular physical automated test cases which can be run to test the application.**

## I. Introduction

TODAY'S efficiency and short time–to–market is the key factor in software development that creates pressure on software development teams and a demand for more efficient methods of software development and testing.

The mission of the development is to deliver the system (or a modification of the system) in an agreed time. In the iterative development it means to fix the issues from the previous cycle, to add new features and to test the system and to verify that every feature of the software works according to the specification. What if a customer suddenly wants a new feature that the developers will have a hard time implementing? Or what if this new functionality affects the entire application, so regression effect rates of the code changes are high and the reliability of previously stable parts of the application is challenged?

Testing in such a cycle is often challenging. Preparation and execution of the tests, if performed manually, requires time which is often not available. This can also affect the accuracy of prepared test cases.

Automation of the test cases is one of the possible ways of how to make the process more efficient.

In this paper we are proposing a model–driven approach to front–end web application testing based on the Interaction Flow Modelling Language (IFML, [3]). We are going to describe the process of transforming the IFML model of the tested application to a set of front–end test cases. The automatic generation without of these tests from the IFML model guarantees their consistency.

In this field, UML [11] is a widely adopted modelling language made to visualize the design of the system. Nevertheless, UML does not capture all aspects of the application. One area where UML is lacking vocabulary and tools is in the modelling of the user interface and interaction. To overcome this gap, a Web Modelling Language (WebML [15]) was created introducing visual notations and a methodology for designing complex data–intensive Web applications. This language later evolved into IFML to cover a wider spectrum of front–end interfaces and the data flows between the application front–end components. IFML was later adopted by the Object Management Group (OMG, [16]) as an industrial standard.

## II. Problem description

During the development stage, changes are frequently made to the web application code base. On the front–end the page layout can change, input elements are added or removed, data–flow of the pages is modified. All of these changes must be tested in order to prove that no error was introduced and that everything works as expected. Without the model-driven approach, both the code of the application and the tests are created manually. Every change made to the code must be synchronized with the tests, so that the tests are testing new functionality with new input elements and new corner case input values. When an element is added to a form, all functional tests associated with that form must be modified accordingly. This maintenance of the automated test scripts causes significant overhead in the development of the software project.

## III. Potential of IFML for test case generation

Data driven application front–end is usually built using reusable components (forms, list views, detail views, etc.). These components have expected behaviour. For example, forms are placed on the page to be filled in with data and sent to the server, lists show record details for the user to view or allow him or her to select one or more records and perform

©2015, PTI

an action on these. All of these operations can be modelled using the IFML notation (see an example in Figure 1).

With precise models and proper tooling (for instance IBM Rational Software Architect, Enterprise Architect, AndroMDA) these models can be transformed to code, different models or just to generate the system scaffolding. Developing an application with tens of screens with various components (forms, list views, etc.) can be a lengthy and repetitive task — every form and every list view must be manually created. This process leads to copy paste style of programming and any possible defects can be easily cloned and introduced many times in the application. Even with the use of a user interface component framework this can be a problem. The efficiency of the process can be increased by the generation of the user interface (UI) from the model [4, 5]. While this is efficient — it is certainly easier to create a model of a number of components and their interaction than to implement them physically — it still does not prove that the application is error free and can be delivered to the users. And in most of the cases, we don't have resources to test manually every screen in the application whenever there's a code change.

A similar approach can be also used to generate test case scenarios — we know how to test the basic functionality of a component and what the code for such a test should look like. For this reason it is highly recommended to use the IFML model originally used for the front–end code generation to generate test case scenario code just by using a different template.

## IV. Related work

Although WebML has been used for more than ten years, IFML is relatively new and was recently standardized. The first applications of the standard are emerging, for instance, a systematic model–driven reverse engineering process to generate an IFML representation from such applications is presented in [17].

IFML notation can be easily extended by adding new containers, components, events or by applying custom UML stereotypes to them as described in [4]. The authors added new components and events (swipe, camera event, location sensor event) to be able to describe mobile specific interfaces and interaction.

IFML represents a prospective modelling tool to describe application front–end and a flexible and easily extensible notation. Hence, we decided to use its capabilities to generate front–end test case scenarios.

In the previous approaches, a general purpose modelling languages, such as UML, that described the system high–level model were often used for system code generation. UML models have normally been used for the process of automated code generation from the model, for example [12, 1, 13]. The same applies for the generation of test cases from the tested application model.

From the previous approaches, sequence diagrams [2, 19], state chart [10] or activity diagrams [9, 14] are used, but these diagrams and the models they represent are more focused on describing the application structure or data flow not the user front–end interaction. In [19] sequence diagrams as the most suitable for precise and detailed description of a system's actions and behaviour. UML notation was also used to generate the user interface. In [5] a new diagram called user interface diagram was introduced with a user interface specialization.

Our goal is to use the approach of generating the test cases from the application model, but instead of UML, which is already covered in the previous work, we are going to generate automated test cases from the IFML model. This area is currently lacking sufficient theoretical support since IFML is a quite new modelling language. In [4] the proposals were verified using manual testing by testers.

## V. Proposed solution

Our proposal of generating automated end–to–end test case scenarios from IFML model is based on the set of transformations, which are outlined in Figure 2.

In the proposed solution, the XML representation of an IFML model is converted into a front–end model using a predefined set of rules. The model is then used to generate abstract test case scenarios. Physical details are added to these scenarios by templates and a set of executable test case scenarios are created. Details of this process are following further on.

## VI. Front–end model

In this section we present details of the front–end model, used for the transformation process introduced above. The aim of this model is to formalize the user interaction with the application.

We consider this model more suitable for generation of the test cases, as the IFML notation is too rich and descriptive for our use. In this proposal, we have used an already defined and published formal model, verified in our previous work [7, 6].

We define a view window $W$ as a set of view containers and a view container $K$ as a tuple $\langle C, N, A, E, D, M \rangle$, where

- $C$ is a hierarchical set of view components placed into this view container (components can be nested)
- $N$ is a set of navigation flows defined as $N : C \cup E \cup D \cup B \rightarrow C \cup A$. The user triggers an event $E$ on view component $C$ with data–bound by $D$ resulting in displaying another view component (or the same one) or triggering an action $A$
- $A$ is a set of actions executed prior to updating the state of the user interface
- $E$ is a set of events a view container and a view component is associated with, the effect of event is the interaction flow
- $B$ is a set of data bound variables whose values will be used in navigation flow
- $D$ is set of data binding expressions d defined as $d : C \rightarrow B$, these expressions extract a value from a view component, for example it describes how to get the numeric value from a text input field

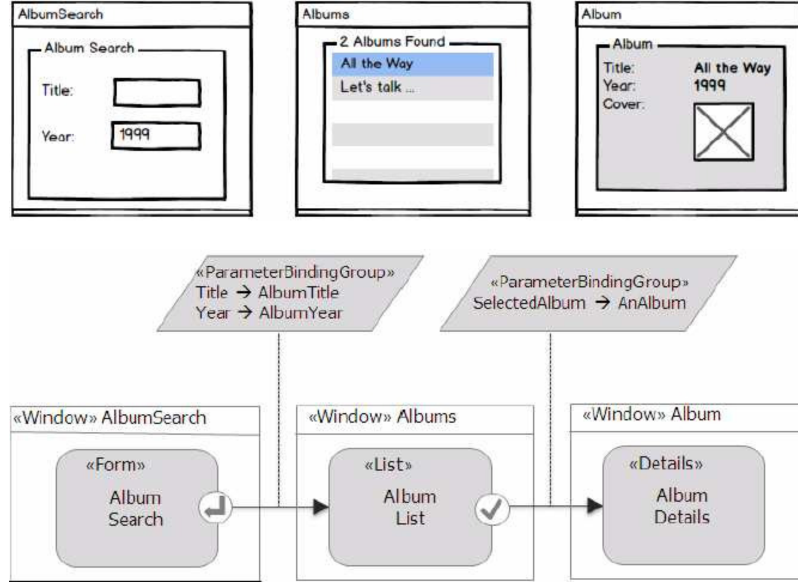


Fig. 1. Example of an IFML model of a "Search for an album" use case

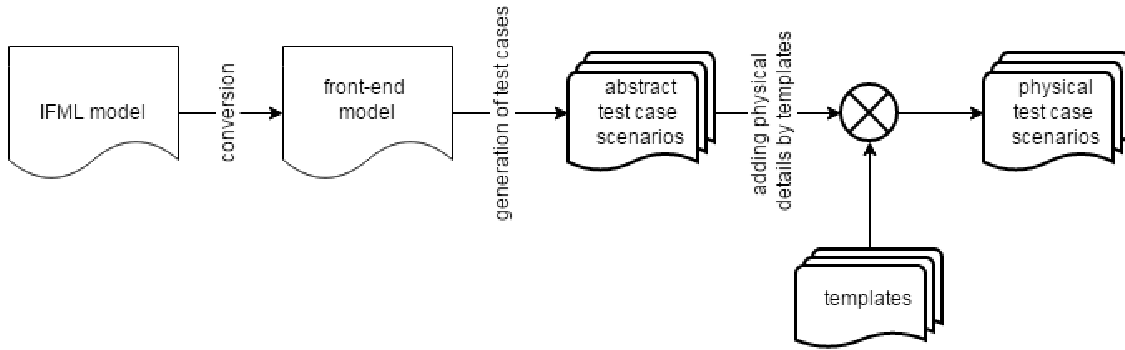


Fig. 2. Transformation of an IFML model to physical automated test case scenarios

- $M$ is a set of custom metadata m to describe the meta–properties of the view components defined as $m : C \rightarrow GM$, where $GM$ is an extensible global set of model meta–properties holding IFML model properties, UML stereotypes and other custom properties

In our example introduced above, the front–end model for the view window from our album search example can be described as the following (we have omitted the definition of $K_{Albums}$ for brevity): $W = \{K_{AlbumsSearch}, K_{Albums}\}$

$K_{AlbumsSearch} = \{$
$C = \{AlbumSearchForm = \{AlbumTitle, AlbumYear\}\}$
$N = \{AlbumSearchForm \cup Submit \cup \{Title, Year\} \rightarrow Albums\}$
$A = \emptyset$
$E = \{Submit\}$
$D = \{AlbumTitle \rightarrow Title, AlbumYear \rightarrow Year\}$,
$B = \{Title, Year\}$,
$M = \{AlbumSearchForm \rightarrow \{Form\}, AlbumTitle \rightarrow \{SimpleField, String\}$,
$AlbumYear \rightarrow \{SimpleField, Year\}\}$
$\}$

The abstract test case scenario $T$ is defined as $T : N \cup C \cup V \rightarrow 0, 1$, $N$ and $C$ are defined as before and $V$ is a set of rules that all has to be matched for the test not to be marked as failed.

For simplicity just assume that when the search operation has finished, the albums view container is displayed (no matter how many results it will show). In that case the result of $T$ is success when $C$ is $AlbumList$:

$$T = 1 \Leftrightarrow V = \{Equals(C, AlbumList)\}$$

For our purposes the front–end model is serialized into JSON format, which is easy to read and can be injected directly into the template engine.

## VII. THE TRANSFORMATION PROCESS

As we have already introduced, the first step of the process is the transformation of an IFML model into a front–end model. Modelling tools use XML format to persist the models

(see Listing 1), however this format is quite heavy for further processing. We have decided to transform the XML format into more readable and lightweight JSON form. For this conversion process, we have defined a set of rules.

```
<interactionFlowModelElements name="AlbumList" xsi:
    type="ext:IFMLWindow">
 <viewElements xsi:type="ext:Form"
   name="AlbumSearchForm">
  <viewElementEvents xsi:type="ext:OnSubmitEvent"
      name="Search">
   <outInteractionFlows>
    <parameterBindingGroup .../>
   </outInteractionFlows>
  </viewElementEvents>
  <viewComponentParts xsi:type="core:DataBinding"
      name="Album"/>
  <viewComponentParts xsi:type="ext:SimpleField"
      name="Title"/>
  <viewComponentParts xsi:type="ext:SimpleField"
      name="Year"/>
 </viewElements>
</interactionFlowModelElements>
```

Listing 1. XML representation of an IFML model

For each IFML element in order to be used in our model a rule is defined. For example for an IFML element with the name “viewElements” or “viewComponentParts” an entry is added to the “components” array, for every “xsi:type” attribute with the value ”ext:Form” a “type: form” entry is added to the “metadata”. In order to track the original elements we also copy the element id that uniquely identifies it. The rule is a simple JavaScript method matching properties of an XML element. For our example presented in Figure 1, the respective XML notation of the IFML model is given in Listing 1 and the JSON description of the front–end model in Listing 2.

```
{
  "name": "AlbumList",
  "metadata": [{ "type": "window" }],
  "components": [{
    "name": "AlbumSearchForm",
    "metadata": [{ "type": "form" }],
    "variables": [{
      "name": "Year",
      "metadata": [{
        "dataType": "int",
        "constraint": "not-empty"
      }]
    },{
      "name": "Title",
      "metadata": [{
        "dataType": "int",
        "constraint": "year"
      }]
    }],
    "binding": [{
      "name": "Title",
      "from": "AlbumTitle"
    },{
      "name": "Year",
      "from": "AlbumYear"
    }],
    "events": [{
      "type": "submit",
      "target": "Albums"
    }],
    "components": [{
      "name": "AlbumTitle",
      "metadata": [{ "type": "field" }]
    },{
      "name": "AlbumYear",
      "metadata": [{ "type": "field" }]
    }]
  }]
}
```

Listing 2. XML representation of an IFML model

In the next step the front–end model is transformed into a set of abstract test case scenarios. These scenarios are independent of the specific technological platform or programming language used to implement it — using the abstracted test cases gives us flexibility to generate the test case in different scripting languages and test automation APIs. From a technical perspective the JSON representation of the front–end model is transformed with the use of predefined transformation rules to JSON representation of the abstract test case scenario (see Listing 3).

```
{
  "name": "AlbumSearchForm",
  "id": "65de40fd-8283",
  "specs": [
    {
      "name": "Search",
      "type": "search",
      "steps": [
        "fill": {
          "locator": "Year",
          "type": "year"
        },
        "submit": { "locator": "AlbumSearchForm" },
        "waitFor": { "locator": "AlbumList" }
      ]
    },
    {"name":"Reset" ...}
  ]
}
```

Listing 3. Abstract test case scenario

In the last step the abstract test case scenarios are transformed into executable (platform specific or programming language specific) test case scenarios. In this process a template system is used to generate the test cases. The template system chooses the desired template from the templates library (WebdriverIO [21] template was used to generate code for our example). The template system is based on the JavaScript Underscore template [18] capability. Abstract test case scenario serialized into JSON format is supplied as a parameter and then processed by the template engine (see Listing 4).

```
var webdriverio = require('../index');
var templates = require('/templates');

describe('<%=scenario.name%>', function() {
  var client = {};
  jasmine.DEFAULT_TIMEOUT_INTERVAL = 9999999;

  beforeEach(function() {
    client = webdriverio.remote({
      desiredCapabilities:{
        browserName: 'phantomjs'}
    });
    client.init();
  });

<% _.each(scenario.specs, function(spec) { %>
  <% var specTemplate = specTemplates.getTemplate(
```

```
      spec.type); %>
  it('<%=toSpecName(spec.name)%>', function(done) {
    <%=specTemplate.renderTemplate(spec); %>
  });
<%});%>

  afterEach(function(done) { client.end(done); });
});
```

Listing 4. Underscore template

In Listing 5 we present the result of the transformation for example from Figure 1. The executable JavaScript is code created to be executed by Jasmine [8] test runner. In the example, we test the basic functionality of a search form. The test describes 'AlbumSearchForm' test suite with a spec 'should search' (spec is named set of expectations to be met). The spec function makes call to a browser automation tool Selenium WebDriver [18] via binding library WebdriverIO. If we want to use different programming language to implement the test cases, a different template can be used to generate physical test case scenarios from abstract test case scenarios.

```
var webdriverio = require('../index');
var templates = require('/templates');

describe('AlbumSearchForm', function() {
  var client = {};
  jasmine.DEFAULT_TIMEOUT_INTERVAL = 9999999;
  beforeEach(function() {
    client = webdriverio.remote({
      desiredCapabilities:{
        browserName: 'phantomjs'}
    });
    client.init();
  });

  it('should search', function(done) {
    client.url('...')
      .setValue('#Year', '2015')
      .submitForm('#AlbumSearchForm')
      .waitForExist('#AlbumList', 2000);
  });

  afterEach(function(done) { client.end(done); });
});
```

Listing 5. Code generated for AlbumSearchForm form from the IFML model (some code left for brevity)

The code first initializes the Selenium web driver to be used with windowless browser called PhantomJS, then navigates to our application page, fills value 2015 into the Year input field, submits the form and waits for AlbumList element to appear. If the element is not displayed within 2 seconds, the test fails.

## VIII. Verification

The proposed solution is currently in the implementation stage with the first results arising from experiments. We have collected the initial feedback from the users of our prototype and adjusted the model accordingly. We have been experimenting with a set of IFML models created for 3 applications we created for our needs and the results are promising. The test case scenarios were correctly generated, but in some specific cases, the generation process should be improved upon further, which nevertheless represents an implementation task. As mentioned before the added value is the automatic generation of test cases when any change is made to the IFML model.

Our solution quickly discovered problems in tested applications in the cases when

- changes were made to existing server–side code introducing an error in processing the client data,
- changes were made to existing client side code introducing an error in JavaScript components,
- new elements were added with faulty behaviour or
- new elements were added, but the values of these components were not handled properly when sent to server.

On several occasions, we ended up in a situation where some tests were generated syntactically correctly but the semantics of expected result assertions was not corresponding to the state of the tested application. This would often lead to the IFML model, a conversion rule or a used template being updated. It was a means of feedback and an indication of how to evolve the solution to be fully functional. Only occasionally we have had to remove an IFML feature so that the test suite could be generated.

The biggest advantage of the use of an IFML model instead of an UML model as a base model for test case generation is the power of IFML to describe the front–end views, components and interaction between them. The test cases are therefore potentially more consistent.

## IX. Conclusions and future work

IFML is a relatively new notation recently standardized by OMG. So far it has not been widely adopted; currently there are only 2 tools on the market (commercial WebRatio [22] and open source Eclipse plugin [20]). The primary target of IFML is to express the content, user interaction and control behaviour of the front–end of software applications, which are some of the key aspects of the application.

The first results of experiments with our proposed solution show that using an IFML model for test case generation is viable and promising. Initial feedback from experiments with our solution confirms an advantage of IFML — it does describe the front–end directly which gives relative high level of assurance about the precision of the generated test case scenarios and their ability to be executed.

In the future work, we are going to improve and extend the proposed solution to support more IFML constructs and be able to help in large–scale development projects. Our solution does not currently support any components other than forms and list views, but new UML stereotype can be added, or the IFML notation can be extended by the addition of a new component to change the transformation and generate new types of test case scenario.

## Acknowledgment

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


[1] Manoli Albert et al. "Automatic generation of basic behavior schemas from UML class diagrams". English. In: *Software & Systems Modeling* 9.1 (2010), pp. 47–67. ISSN: 1619-1366. DOI: 10.1007/s10270-008-0108-x. URL: http://dx.doi.org/10.1007/s10270-008-0108-x.

[2] A. Bandyopadhyay and S. Ghosh. "Test Input Generation Using UML Sequence and State Machines Models". In: *Software Testing Verification and Validation, 2009. ICST '09. International Conference on*. Apr. 2009, pp. 121–130. DOI: 10.1109/ICST.2009.23.

[3] Marco Brambilla and Piero Fraternali. *Interaction Flow Modeling Language: Model-Driven UI Engineering of Web and Mobile Apps with IFML*. Morgan Kaufmann, 2014.

[4] Marco Brambilla, Andrea Mauri, and Eric Umuhoza. "Extending the Interaction Flow Modeling Language (IFML) for Model Driven Development of Mobile Applications Front End". English. In: *Mobile Web Information Systems*. Ed. by Irfan Awan et al. Vol. 8640. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 176–191. ISBN: 978-3-319-10358-7. DOI: 10.1007/978-3-319-10359-4_15. URL: http://dx.doi.org/10.1007/978-3-319-10359-4_15.

[5] F. Ferri. *Visual Languages for Interactive Computing: Definitions and Formalizations*. Premier reference source. Information Science Reference, 2008. ISBN: 9781599045368. URL: https://books.google.co.uk/books?id=LNOSq-q7wfoC.

[6] Karel Frajták, Miroslav Bureš, and Ivan Jelínek. "Formal specification to support advanced model based testing". In: *Computer Science and Information Systems (FedCSIS), 2012 Federated Conference on*. IEEE. 2012, pp. 1311–1314.

[7] Karel Frajták, Miroslav Bureš, and Ivan Jelínek. "Manual testing of web software systems supported by direct guidance of the tester based on design model". In: *World Academy of Science, Engineering and Technology* (2011), pp. 542–545.

[8] *Jasmine, behavior-driven development framework for testing JavaScript code @ONLINE*. http://jasmine.github.io.

[9] S. Kansomkeat, P. Thiket, and J. Offutt. "Generating test cases from UML activity diagrams using the Condition-Classification Tree Method". In: *Software Technology and Engineering (ICSTE), 2010 2nd International Conference on*. Vol. 1. Oct. 2010, DOI: 10.1109/ICSTE.2010.5608913.

[10] Supaporn Kansomkeat and Wanchai Rivepiboon. "Automated-generating Test Case Using UML Statechart Diagrams". In: *Proceedings of the 2003 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on Enablement Through Technology*. SAICSIT '03. Johannesburg, South Africa: South African Institute for Computer Scientists and Information Technologists, 2003, pp. 296–300. ISBN: 1-58113-774-5. URL: http://dl.acm.org/citation.cfm?id=954014.954046.

[11] Andrey Karpov. *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. Nov. 2011.

[12] D. Kundu, D. Samanta, and R. Mall. "Automatic code generation from unified modelling language sequence diagrams". In: *Software, IET* 7.1 (Feb. 2013), pp. 12–28. ISSN: 1751-8806. DOI: 10.1049/iet-sen.2011.0080.

[13] Abid Mehmood and Dayang N.A. Jawawi. "Aspect-oriented model-driven code generation: A systematic mapping study". In: *Information and Software Technology* 55.2 (2013). Special Section: Component-Based Software Engineering (CBSE), 2011, pp. 395–411. ISSN: 0950-5849. DOI: http://dx.doi.org/10.1016/j.infsof.2012.09.003. URL: http://www.sciencedirect.com/science/article/pii/S0950584912001863.

[14] Chen Mingsong, Qiu Xiaokang, and Li Xuandong. "Automatic Test Case Generation for UML Activity Diagrams". In: *Proceedings of the 2006 International Workshop on Automation of Software Test*. AST '06. Shanghai, China: ACM, 2006, pp. 2–8. ISBN: 1-59593-408-1. DOI: 10.1145/1138929.1138931. URL: http://doi.acm.org/10.1145/1138929.1138931.

[15] N. Moreno, P. Fraternali, and Antonio Vallecillo. "WebML modelling in UML". In: *Software, IET* 1.3 (June 2007), pp. 67–80. ISSN: 1751-8806.

[16] *Object Management Group @ONLINE*. http:///www.omg.org.

[17] Roberto Rodriguez-Echeverria et al. "IFML-based Model-Driven Front-End Modernization". In: (2014).

[18] *Selenium, web browser automation @ONLINE*. http://www.seleniumhq.org.

[19] Aristos Stavrou and GeorgeA. Papadopoulos. "Automatic Generation of Executable Code from Software Architecture Models". English. In: *Information Systems Development*. Ed. by Chris Barry et al. Springer US, 2009, pp. 1047–1058. ISBN: 978-0-387-78577-6. DOI: 10.1007/978-0-387-78578-3_36. URL: http://dx.doi.org/10.1007/978-0-387-78578-3_36.

[20] *The open source IFML editor - Based on Sirius @ONLINE*. https://github.com/ifml/ifml-editor.

[21] *WebdriverIO, Selenium 2.0 bindings for NodeJS @ONLINE*. http://webdriver.io.

[22] *WebRatio @ONLINE*. http://www.webratio.io.