FITTEST: A New Continuous and Automated Testing Process for Future Internet Applications

Tanja Vos
Universidad Politécnica de Valencia, Spain
tvos@pros.upv.es

Paolo Tonella
Fondazione Bruno Kessler, Trento, Italy
tonella@fbk.eu

Wishnu Prasetya
Universiteit van Utrecht, Utrecht, Netherlands
wishnu@uu.nl

Peter M. Kruse
Berner & Mattner, Berlin, Germany
Peter.Kruse@berner-mattner.com

Alessandra Bagnato
SOFTEAM, Paris, France
alebagnato@softeam.fr

Mark Harman
University College London, London, UK
mark.harman@ucl.ac.uk

Onn Shehory
IBM Research Haifa, Haifa, Israel
on@il.ibm.com

Abstract—Since our society is becoming increasingly dependent on applications emerging on the Future Internet, quality of these applications becomes a matter that cannot be neglected. However, the complexity of the technologies involved in Future Internet applications makes testing extremely challenging. The EU FP7 FITTEST project has addressed some of these challenges by developing and evaluating a Continuous and Integrated Testing Environment that monitors a Future Internet application when it runs such that it can automatically adapt the testware to the dynamically changing behaviour of the application.

I. INTRODUCTION

Testwares developed for traditional software have to be evolved when the application changes. Depending on the kind of change, new test cases are added, some are removed and some are adapted to the changed functionalities. Regression testing is also conducted, to ensure that test cases that are still executable continue to satisfy their oracles. This evolution of testwares becomes a continuous need for Future Internet (FI) applications, since these applications evolve at an extreme rate, due to the self-adaptive and dynamic code loading mechanisms they rely upon.

For FI applications, self-modifications, context and environment dependent autonomous behaviours, as well as user defined customisation and dynamic re-configurations might have gone untested before the release of the software. Because of this, the testwares might be already inadequate for a specific executing instance of the system. Then, services and components could be dynamically added by customers and the intended use could change significantly. Therefore, the testwares need to evolve automatically together with the system.

Goal of the FITTEST (Future Internet Testing) European project (see Table I) is to develop a novel FI testing approach, in which testing is performed continuously, even after deployment to the customer. This is unlike the traditional approach, where testing is done before the deployment of a major release. Behavioural models and oracles are inferred from logged executions, and adequate test cases are generated from the models on demand. The FITTEST tools are aware of the state of the System Under Test (SUT) and support dynamic runtime classification for combinatorial testing, and online test case generation and execution.

This paper presents the main project outcome, i.e., the FITTEST Continuous and Integrated Testing Environment (ITE), which can monitor the FI application when it runs and can adapt the testware to the dynamic changes observed, so as to continuously test the changing behaviour. By relying on various inference techniques, the ITE tests the SUT against previous instances of itself, by comparing fresh executions (produced by test-cases generated from inferred models), against inferred, automated oracles (aka anomaly detectors) [13].

II. FITTEST ITE AND ITS CONTINUOUS CYCLE

FITTEST ITE is a distributed test environment. ITE’s main component runs at the tester’s workstation and is in charge of the inference of models and oracles, and the generation of test cases. Through agents, it monitors and collects data from the SUT running in its production environment. The SUT typically consists of a server part (typically running a Web server) and users’ clients, typically running the application within a Web browser. ITE’s agents are deployed to monitor them. To execute the generated test-cases the ITE also controls

TABLE I

<table>
<thead>
<tr>
<th>Name and acronym</th>
<th>Future Internet Testing (FITTEST)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EU call and project ID</td>
<td>ICT-2009.1.2, ID n. 257574</td>
</tr>
<tr>
<td>Dates</td>
<td>September 1, 2010 – December 31, 2013</td>
</tr>
<tr>
<td>Source of funding</td>
<td>European Commission</td>
</tr>
<tr>
<td>Total budget; EU contrib.</td>
<td>5.8M EUR; 3.9M EUR</td>
</tr>
<tr>
<td>Project consortium</td>
<td>Universidad Politécnica de Valencia, Spain</td>
</tr>
<tr>
<td></td>
<td>University College London, UK</td>
</tr>
<tr>
<td></td>
<td>IBM research Haifa, Israel</td>
</tr>
<tr>
<td>Web site</td>
<td><a href="http://crest.cs.ucl.ac.uk/fittest">http://crest.cs.ucl.ac.uk/fittest</a></td>
</tr>
</tbody>
</table>
Fig. 1. The FITTEST Continuous Testing Cycle

The format allows deep objects to be serialized into the logs. In the Flash logger, we can register serialization delegates. They specify how instances of the classes they delegate are to be serialized, e.g., we may want to just log some fields, or to first apply some summation function over them. Note that using such a delegate we can also specify an abstract projection of the application’s concrete state, and log it. Although the delegates have to be manually specified, they are 'transparent': they do not clutter the application; the application is not even aware them.

The PHP logger does not currently log state information, hence models inferred from PHP logs are purely event-based.

For Flash, the ITE also includes a bytecode instrumentation tool called Abci [11], to instrument selected methods and instruction-blocks for logging, thus allowing much more detailed logs to be produced. Abci is implemented with the AG attribute grammar framework [16], allowing future extensions.

B. Model Inference

Model inference is a key activity in continuous testing [5]. In general, it uses execution traces and infers models that describe structure and behaviour of a SUT using either event-sequence abstraction or state abstraction [9], [3], [4]. One of the most frequently used format of model is the Finite State Machine (FSM). In a FSM, nodes represent states of the SUT, and transitions represent an application event/action (e.g., a method call, an event handler) that can change the application state, if executed. Additionally, guards and conditions can enrich the model to capture the context in which events and actions are executed.

In the ITE, models are automatically inferred from logs using either event-sequence abstraction or state abstraction. The outcome are FSM models that can be fed to the successive phases of the ITE cycle (specifically, event sequence generation and input data generation). Models are updated continuously and incrementally [10] as the software evolves and new behaviours appear in the execution logs. Figure 2

III. THE FITTEST TECHNOLOGIES AND TOOLS

A. Logging and Instrumentation

Loggers automatically intercept events representing top level interactions with the application under test (this way programmers do not need to explicitly write any logging code). The used technology determines what constitutes top level interactions. The ITE currently supports Flash and PHP. For a PHP application, top level interactions are the HTTP requests sent by the client to the application. For a Flash application, these are GUI events, such as the user filling a textfield or clicking on a button. The produced logs are in the compact but deeply structured FITTEST format [14]. FITTEST logs are about twice smaller than XML, and can be even smaller if we apply compression. The runtime overhead imposed by our logger was acceptable in all use cases conducted so far within FITTEST.
shows an example of FSM model, it has 5 states and 4 different events.

The events in the model often have input parameters. From the log we extract concrete input values for such parameters. We have implemented also a data mining technique to infer input classes (for combinatorial testing, used later in test generation) for these parameters.

C. Oracle Inference

The ITE uses Daikon [7] to infer oracles from the logs, in the form of pre- and post-conditions for each type of high level event. In Figure 1 these are called property-based oracles. Figure 3 shows some examples of pre- and post-conditions inferred for the event change on a slider GUI component named priceSlider. E.g., the post-condition says that the slider does not change the number of items in the shopping cart. Since automatically inferred invariants are “likely” invariants, manual oracle validation is required to filter out incorrect oracle candidates.

<table>
<thead>
<tr>
<th>priceSlider</th>
<th>change</th>
<th>(pre)</th>
<th>arg1 ≠ arg2</th>
<th>34×</th>
<th>cartTotal = old(cartTotal)</th>
<th>34×</th>
</tr>
</thead>
</table>

If the logged (abstract) state consists of many variables, simply passing them all to Daikon will cause Daikon to try to infer all sorts of relations between them, most of which are actually not interesting for us. The ITE allows groups of variables to be specified (using regular expressions) and constrains Daikon to only infer the oracles for each group separately. In principle, it would then be possible to control the ITE to infer, e.g., pair-wise oracles.

The ITE also infers so-called pattern-based oracles [6]. These are algebraic equations of common patterns over high level events, e.g., \( a = \epsilon \) or \( bb = b \), saying that on any state, executing the event \( b \) twice will lead to the same state as executing just one \( b \).

Finally, through instrumentation (Subsection III-A) we can also log the entry and exit points of selected methods, allowing pre- and post-conditions to be inferred for them. Furthermore, so-called sub-cases oracles can also be inferred [15]. This requires “instructions blocks” to be instrumented as well — for each method, transitions between these blocks form the method’s control flow graph. Visit patterns over these blocks can be specified and used as some form of splitters [2], yielding stronger oracles.

D. Test Case Generation

Once a model is available, combinatorial testing or search based testing can be used to generate test cases.

1) Combinatorial-model-based test generation: This method [12] starts from a finite state model and applies model-based testing to generate test paths that represent sequences of events to be executed against the SUT. Several algorithms are used to generate test paths, ranging from simple graph visit algorithms, to advanced techniques based on maximum diversity of the event frequencies, and semantic interactions between successive events in the sequence.

Such paths are, then, transformed to classification trees using the CTE XL\(^1\) format, enriched with domain input classifications such as data types and partitions. Then, test combinations are generated from those trees using t-way combinatorial criteria. Finally, they are transformed to an executable format depending on the target application (e.g., using Selenium\(^2\)). Note that although these test cases are generated from models learned from the SUT’s own behaviour, the combinatorial stage may cause them to trigger fresh behaviour, and thus testing it against the learned patterns. Listing 1 shows an example of a test generated for Flexstore\(^3\), consisting of a sequence of event calls, e.g., itemClick, change, on some GUI items, e.g., ButtonBar0, and the values for the input parameters.

Listing 1. Generated code of a test case in Selenium for Flexstore.

```java
public void test_c56()
{
    driver.invoke("Nokia 6670", "details");
    driver.invoke("series", "change", "4");
    driver.invoke("Button11", "click");
    driver.invoke("series", "change", "2");
    driver.invoke("Button2", "click");
    driver.invoke("series", "change", "4");
    driver.invoke("Nokia 6670", "details");
}
```

Thanks to the integration with CTE XL, other advanced analysis can be performed on the classification trees. For

\(^1\)http://www.cte-xl-professional.com

\(^2\)http://seleniumhq.org

\(^3\)www.adobe.com/devnet/flex/samples/flex_store_v2.html

Fig. 2. Example of inferred FSM for Flexstore, zoomed on state S1

Fig. 3. Examples inferred log-based oracles.
example, we can impose dependencies on the input classifications such as City=Berlin IMPLIES Country=Germany and filter the test combinations that violate the dependencies to remove invalid tests.

2) Search-model-based test generation: In this approach [1] the ITE transforms the inferred models and the conditions that characterize the states of the models into a Java program. It then uses evolutionary, search-based testing, to automatically produce test cases that achieve branch coverage on the Java representation. Specifically, we use Evosuite [8] for this task. Branch coverage solved by Evosuite has been theoretically shown to be equivalent to transition coverage in the original models. Finally, the test cases generated by Evosuite require a straightforward transformation to be usable to test our SUT.

E. Test Evaluation

The generated test-cases are executed and checked against oracles. If an oracle is violated, the corresponding test case has then failed. Further investigation is needed to understand the reason behind the failure. Support for automatic debugging is currently beyond our scope. As depicted in Figure 1, various oracles can be used for test evaluation. The most basic and inexpensive oracles detect SUT crashes; the most effective but expensive are human oracles. The ITE offers several other types of oracles that are automatically inferred from the logs or the models.

1) Model-based oracles: The test cases produced by the ITE are essentially paths generated from the FSM models. An additional property of such a path is that it is expected to be executable: it should not crash or become stuck in the middle, which can be easily checked. In Figure 1 this is called model-based oracles.

2) Property- and pattern-based oracles: The executions of the generated test cases are also logged. The ITE then analyze the produced logs to detect violations to the inferred pre- and post-conditions as well as the pattern-based oracles. The checking of the latter is done at the suite level rather than at the test case level. E.g., to check $bb = b$ the ITE essentially looks for a pair of prefix sequences $\sigma bb$ and $\sigma b$ in the suite that contradicts the equation.

IV. CONCLUSIONS

We have described the workflow and technologies involved in the FITTEST ITE for continuous testing of Future Internet applications. Logging and automated inference of models and oracles lie at the heart of our continuous testing approach as opposed to traditional testing.

The FITTEST technologies have been evaluated through industrial case studies following the FITTEST evaluation framework [17]. We are currently completing the two empirical studies that evaluate the continuous testing cycle. The first study is being carried out in an experimental environment, where the decision of deploying functionality changes in the SUT will be under the control of the experimenter. The SUT that we will use is the aforementioned Flash webshop called Flexstore from Adobe. With this study, we aim to iteratively investigate the automated evolvability of the testware (i.e., models, test cases and oracles) that are generated during our continuous testing approach.

The second empirical study is done with the purpose of scaling-up the complexity of our SUT, as well as investigating additionally the non-obtrusiveness of our approach. This study is carried out in a real environment within the company SOFTEAM4, by testing their Modelio SaaS modelling in the Cloud tool5.

ACKNOWLEDGMENT

This work was financed by the FITTEST project, ICT-2009.1.2 no 257574.

REFERENCES


4http://www.softeam.com/
5http://www.modeliosoft.com/