FITTEST: a new continuous and automated testing process for future internet applications

Tanja Vos
Universidad Politécnica de Valencia
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 Harmann
University College London
London, UK
mark.harman@ucl.ac.uk

Onn Shehory
IBM research Haifa
Haifa, Israel
onn@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.**

Keywords*—*Future Internet, Quality, Continuous Testing, Test Automation.

I. Introduction

Testware developed for traditional software has 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 in an extreme rate, due to the self-adaptive and dynamic code loading mechanisms they rely upon.

For FI applications, the testwares might even be already inadequate for a specific executing instance of the system, because of the self-modifications, context and environment dependent autonomous behaviours, as well as user defined customisation and dynamic re-configurations, which might have gone untested during the testing phase that precedes the release of the software. Services and components could be dynamically added by customers and the intended use could change significantly. Therefore, the testwares for FI systems need to evolve automatically together with the system at an unprecedented rate.

In our approach, 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. Our tools are aware of the current state of the System Under Test (SUT), supporting dynamic runtime classification for combinatorial testing, and online test case generation and execution.

This paper presents an high-level overview of our 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. This way, FITTEST departs from the traditional approach in which we assume to know which values are valid input values for the SUT, and what the expected responses should be. Confronted with a highly dynamic and context-aware SUT, we cannot assume any of these. Consequently, by relying on various inference techniques, we now essentially test 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]. This "looking-in-the-mirror" approach is a valid testing approach, provided the false positive rate is low enough, with the advantage that it can be completely automated and it can easily evolve along with the SUT.

Paper overview: Section II describes the ITE and the steps and flows that constitute its continuous testing cycle, Section III briefly presents the technologies implemented in the tool and finally Section IV concludes the paper.

II. FITTEST ITE and its continuous cycle

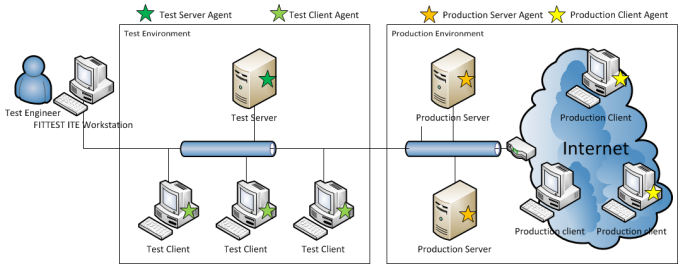

Fig. 1. The FITTEST ITE testing environment

FITTEST ITE is a distributed test environment. When deployed, it typically looks as in Figure 1. ITE's main component runs at the tester's workstation and is in charge of e.g. 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 a replica of the SUT that is running in a dedicated testing environment.

A continuous testing cycle using the ITE consists of the steps shown in Figure 2. The ITE activates the logging of the SUT, and then collects the generated logs. We can either log real usages by end users of the application in the production environment, or log test cases execution in the test environment.

The ITE analyses the logs and infers a model of the SUT. Whenever a model change is detected, the ITE also initiates oracles inference (e.g. log-based or model-based oracles), test cases generation, and finally test cases execution. To generate test cases, abstract test cases are first inferred from the model by traversing the model according to a proper criteria, such as transition coverage. Then, they are concretised by means of search-based test case generation (using a genetic algorithm).

The ITE executes the test cases and evaluates them using free oracles (e.g. SUT crashes or SUT hangs) and inferred oracles. Of course, it is also possible to incorporate human written oracles.

The implementation of the tools for logging and test case execution depend on the technologies used by the SUT. Currently the ITE supports Flash and PHP applications.

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. As indicated above, 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 are about twice smaller than XML, and can be even smaller if we apply compression.

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.

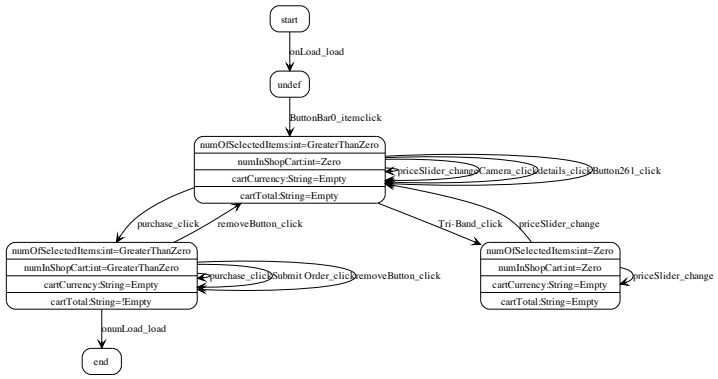

Fig. 3. An example of an inferred FSM model for Flexstore.

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 [17], allowing future extensions, such as adding more instrumentation types, to be programmed more abstractly.

B. Model Inference

Model inference is a key activity in continuous testing [5], [16]. 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 3 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 2 these are called *property-based oracles*. Figure 4 shows some examples of pre- and post-conditions

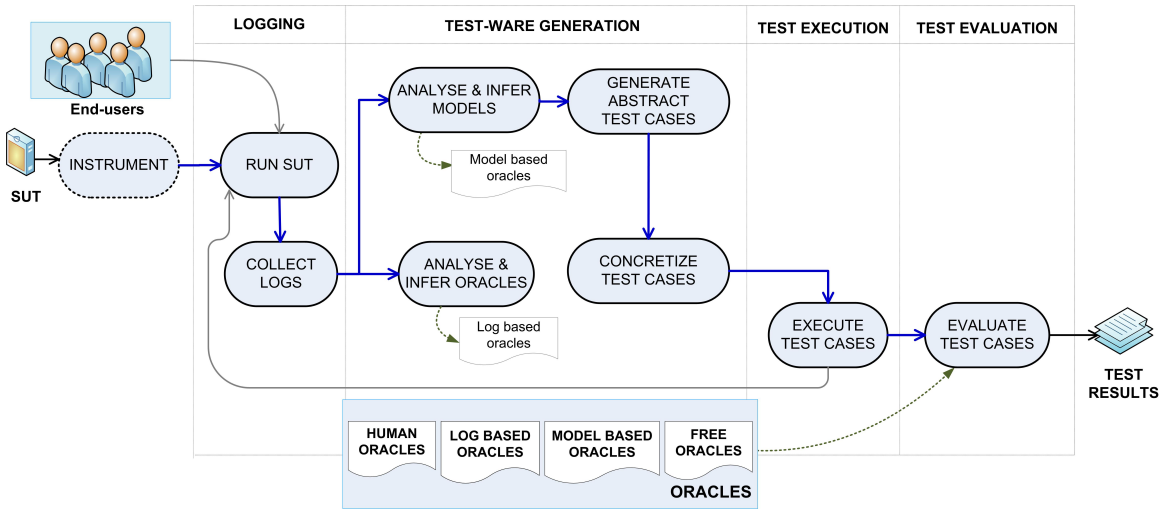

Fig. 2. The FITTEST Continuous Testing Cycle

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.

priceSlider			
change	(pre)	$arg_0 \neq arg_1$	34×
	(post)	$cartTotal = \mathbf{old}(cartTotal)$	34×
		...	

Fig. 4. Examples inferred log-based oracles.

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 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.

```
public void test_c56(){
        driver.invoke("ButtonBar0", "itemclick", "1");
        driver.invoke("series", "change", "4");
        driver.invoke("series", "change", "2");
        driver.invoke("Button11", "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 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

[1] http://www.cte-xl-professional.com
[2] http://seleniumhq.org
[3] www.adobe.com/devnet/flex/samples/flex_store_v2.html

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 2, 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 2 this is called *model-based oracles*.

2) Property- and pattern-based oracles: The executions of the 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 σbb and σb 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. As can be seen from the description, logging and automated inference of models and oracles lie at the heart of our continuous testing approach as opposed to the traditional way of testing.

The FITTEST technologies have been evaluated through industrial case studies following the FITTEST evaluation framework for testing tools that can be found here [18].

We are currently finishing up, the execution of the last two empirical studies for evaluating 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 SOFTEAM[4], by testing their Modelio SaaS modelling in the Cloud tool[5].

Acknowledgment

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

References

[1] FITTEST deliverable D4.3: Test data generation and UML2 profile.

[2] The daikon invariant detector user manual. groups.csail.mit.edu/pag/daikon/download/doc/daikon.html, 2010.

[3] A. Babenko, L. Mariani, and F. Pastore. AVA: automated interpretation of dynamically detected anomalies. In *proceedings of the International Symposium on Software Testing and Analysis*, 2009.

[4] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with ADABU. In *proceedings of the International Workshop on Dynamic Systems Analysis*, 2006.

[5] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos. A survey on model-based testing approaches: a systematic review. In *1st ACM Int. ws. on Empirical assessment of software engineering languages and technologies*, pages 31–36, NY, USA, 2007. ACM.

[6] A. Elyasov, I. Prasetya, and J. Hage. Guided algebraic specification mining for failure simplification. In *accepted in the 25th IFIP International Conference on Testing Software and System (ICTSS)*, 2013.

[7] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, pages 35–45, 2007.

[8] G. Fraser and A. Arcuri. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 13th conference on Foundations of Software Engineering*, ESEC/FSE, pages 416–419, New York, NY, USA, 2011. ACM.

[9] A. Marchetto, P. Tonella, and F. Ricca. Reajax: a reverse engineering tool for ajax web applications. *Software, IET*, 6(1):33–49, 2012.

[10] L. Mariani, A. Marchetto, C. D. Nguyen, P. Tonella, and A. I. Baars. Revolution: Automatic evolution of mined specifications. In *ISSRE*, pages 241–250, 2012.

[11] A. Middelkoop, A. Elyasov, and I. Prasetya. Functional instrumentation of actionscript programs with asil. In *Proc. of the Symp. on Implementation and Application of Functional Languages (IFL)*, volume 7257 of *LNCS*. 2012.

[12] C. D. Nguyen, A. Marchetto, and P. Tonella. Combining model-based and combinatorial testing for effective test case generation. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 100–110. ACM, 2012.

[13] C. D. Nguyen, A. Marchetto, and P. Tonella. Automated oracles: an empirical study on cost and effectiveness. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 136–146, New York, NY, USA, 2013. ACM.

[14] I. Prasetya, A. Elyasov, A. Middelkoop, and J. Hage. Fittest log format (version 1.1). Technical Report UUCS-2012-014, Utrecht University, 2012.

[15] I. Prasetya, J. Hage, and A. Elyasov. Using sub-cases to improve log-based oracles inference. Technical Report UUCS-2012-012, Utrecht University, 2012.

[16] M. Shafique and Y. Labiche. A systematic review of model based testing tool support. Technical Report Technical Report SCE-10-04, Carleton University, Canada, May 2010.

[17] S. D. Swierstra et al. UU Attribute Grammar System. http://www.cs.uu.nl/wiki/HUT/AttributeGrammarSystem, 1998.

[18] T. E. J. Vos, B. Marín, M. J. Escalona, and A. Marchetto. A methodological framework for evaluating software testing techniques and tools. In *2012 12th International Conference on Quality Software, Xi'an, Shaanxi, China, August 27-29*, pages 230–239, 2012.

[4] http://www.softeam.com/

[5] http://www.modeliosoft.com/