FITE: Future Integrated Testing Environment *

Michael W. Whalen
Department of Computer Science and Engineering
University of Minnesota
200 Union St. 4-192
Minneapolis, MN 55455
whalen@cs.umn.edu

Patrice Godefroid
Microsoft Research
Redmond, WA, USA
pg@microsoft.com

Leonardo Mariani
Dipartimento di Informatica, Sistemistica e Comunicazione
Università degli Studi di Milano Bicocca
Viale Sarca, 336
Milano – ITALY
mariani@disco.unimib.it

Andrea Polini
Computer Science Division
School of Science and Technologies
Università degli Studi di Camerino
Via Madonna delle Carceri, 9
Camerino (MC) – ITALY
andrea.polini@unicam.it

Nikolai Tillmann
Microsoft Research
Redmond, WA, USA
nikolait@microsoft.com

Willem Visser
Department of Mathematical Sciences
Computer Science Division
University of Stellenbosch
7602 Matieland – SOUTH AFRICA
willem@gmail.com

ABSTRACT

It is well known that the later software errors are discovered during the development process, the more costly they are to repair, yet testing and automated analysis tools tend to be applied late in the development cycle. In this paper, we describe a future integrated testing environment (FITE) that continually analyzes code for a variety of functional and non-functional properties to provide developer feedback as code is being written. This instant feedback allows developers to fix errors as they are introduced, increasing developer productivity and software quality.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments; D.2.5 [**Software Engineering**]: Testing—*symbolic execution, tools*; D.2.4 [**Software Engineering**]: Verification

General Terms

Verification

*This paper is derived from a whitepaper produced for the Dagstuhl Workshop on Practical Aspects of Software Testing, March, 2010

FoSER 2010, November 7–8, 2010, Santa Fe, New Mexico, USA.

Keywords

incremental analysis, compositional analysis, non-functional analysis

1. MOTIVATION AND BACKGROUND

It is well known that the later software errors are discovered during the development process, the more costly they are to repair [5]. Recently, automatic tools based on static and dynamic analysis have become widely used in industry to detect errors, such as null pointer dereferences, array indexing errors, assertion violations, etc [4]. Static analysis techniques have also been used for property verification [11], but scaling issues and the developer effort required to write properties has limited the adoption of these techniques.

These techniques and tools are typically applied late in the development cycle (if at all), and this late application leads to several problems in the adoption and best use of such tools. First, the errors detected at these latter phases of development are expensive to repair. Second, it is difficult to scale tools to analyze an entire large code base at a time. Third, there are also human factor issues that come into play, namely, that the volume of possible false errors over a large code base will overwhelm the user and thus they ignore the results. The inverse occurs as well: tool developers suppress many real errors in an effort to reduce false warnings due to the usage patterns of these tools.

To address these issues we suggest that code should be continuously analyzed starting from an early stage of development, preferably as it is written. This instant feedback will allow developers to repair errors as they are introduced, when it is still cheap to do so. Similar to *continuous integration* with static analysis tools [2] and *continuous testing* [13], the analyses will run in the background on multiple cores or in the cloud, concurrent with development activities. The tools will be based on incremental analysis and cached method summaries [7] that are used to provide pre-

cise analysis and dramatically increase the scale of programs that can be analyzed. This in turn will improve user interface issues, since the increment of code that is analyzed is relatively small, leading to a relatively small number of error messages being displayed to the user at one time. Dynamic analysis, in particular testing, can also benefit since it can use the static analysis results (for the code increment) to produce tests to cover potential errors as well as provide high code coverage.

2. FITE VISION

In order to make our ideas concrete, we propose an "integrated development environment of the future" with testing and analysis playing a prominent, if not central, role. We call this tool the Future Integrated Test Environment (FITE). FITE will continuously test and analyze the increments and will produce recommendations to the user to repair and test the code. To address scaling up from units, FITE will combine incremental analysis with compositional reasoning. Since the tool is based on interaction with a user, human factors will play a large role in the design of FITE. Specifically, the design of FITE should be modular to allow for many different functional and non-functional analyses of interest. In order to not overwhelm the user with too many recommendations from the tool to improve the code and tests, we foresee a pluggable view-based approach, where the user selects the kinds of analysis it wants to perform on the code (such as security, performance, reliability or numerical precision analysis) and the tool produces only recommendations addressing this selection. For example, for 'performance' the tool might show code paths with high worst-case execution times whereas 'security' might focus on buffer overruns and information leaks. This approach is similar to existing static analysis tools, such as Coverity [4], which allow selective enabling and disabling of analyses.

3. FROM DREAM TO REALITY

In order to provide the immediate, concrete feedback necessary for the FITE vision, several research problems must be addressed, most notably in compositional analysis of programs, but also in presenting results to users in a useful but not distracting fashion.

3.1 Compositional Analysis

The key to moving our vision to reality is to effectively engineer compositional reasoning and analysis of large programs, in order to bridge the gap from unit analysis to system analysis, and ultimately the gap between developers and testers.

Two key sub-problems need to be addressed:

1. how to decompose large programs into smaller subcomponents by identifying *interfaces* where those subcomponents can be decoupled [6].
2. next, how to generate *contracts* at those interfaces, in order to capture *input preconditions* and *output postconditions* in the form of constraints that *may happen* or *must hold* [10].

We envision a semi-automated process to solve these two problems of *interface extraction* and *contracts generation*. Those contracts are code *annotations* that capture semantic information about possible behaviors of the program.

Initially, in order to bootstrap the process, a fully-automatic static analysis of the program could first infer candidates interfaces on how to decompose the system (e.g., using heuristics based on measuring the "complexity" of those interfaces) and suggest those to the user. Those interfaces could then be associated with pre-computed contracts of two types: *may contracts* inferred by static analysis (such as "input integer variable x may have any value" or "output return value y may be any integer") and *must contracts* inferred by dynamic analysis of executions obtained with existing or automatically generated test cases (such as "input pointer p must be non-NULL" or "output pointer q always points to an allocated struct of type foo").

Despite this fully-automatic default mode, we really envision a *interactive* (semi-automatic) usage of the FITE tool where the user can be continually involved by receiving and providing feedback. Think of it as *pair programming* where FITE is your coding and test partner who interacts with you as you write code, test it, and explore its possible behaviors. By means of the may and must contracts which can be inserted anywhere in the code (not just at component interfaces), the user and the tool communicate with each other, enriching the raw code with annotations capturing the intent and correctness of the code. The tool also actively suggests annotations by prompting the user (e.g., "do you assume this input pointer is always non-null?" or "did you mean to return a pointer that points to sometime allocated memory (program path A) and sometimes NULL (program path B)?").

Compositional reasoning allows the automatic inference of properties of the whole system by combining properties of sub-units: may contracts (summaries) can be combined to prove that some bad things cannot happen (proofs) while must contracts can be used for automatic test generation of system tests and bug finding. The framework can be extended to functional properties and non-functional properties.

3.2 Non-functional Analysis

Non-functional properties, such as performance, can cause some of the most expensive and difficult to debug problems within applications. However, *which* non-functional properties are important often depends upon the type of application being written. For example, an authentication server is critically concerned with security, while an embedded system may have no security constraints but may require worst-case execution time bounds and guarantees of numeric precision.

The FITE architecture will support plug-ins that can perform a wide variety of specialized, non-functional analysis. The IDE itself will have the concept of an analysis load-set, which allows a developer or project manager to determine which non-functional analysis are available (and most important) for the class of application being created.

We consider a handful of non-functional analysis below. These are meant to be representative rather than exhaustive:

Worst-Case/Average Case Timing Analysis: A standard area of concern for developers are possible performance bottlenecks within an application. Symbolic evaluation tools such as JPF [1] and Pex [14] allow examination of code paths to determine which paths are longest or are known to make "expensive" API calls.

A plug-in that could flag potential performance bottle-

necks could involve predefined configuration data that catalogues the relative cost of system functions and integration of this data with either (1) a symbolic simulator to describe feasible paths through the code or (2) an abstract interpretation engine. The symbolic simulator may have an advantage in that it may be able to sum-across-paths to talk about variance between symbolic paths and approximate average case performance, while abstract interpretation may be better at providing conservative guarantees about worst-case execution time.

Security: Security problems are endemic to modern software. Old problems such as buffer overflows, continue to plague a variety of applications. More generally, attacks involving authentication, back-doors, SQL injection attacks, input validation, and many other causes cost billions of dollars in direct costs (to find, fix and patch) and in indirect costs (e.g., identity theft).

Existing software tools, such as SAGE [9], can automate many buffer overflow checks. SAGE is designed to run on large binary programs. We believe that we can create more precise analysis by reducing the scale of programs to be analyzed, and to broaden the category of attacks that can be checked. For example, statements creating dynamic SQL queries should be flagged and analyzed to prevent SQL injection attacks. FITE should both generate test cases that demonstrate SQL-injection attacks and also provide automated suggestions for writing SQL-injection resistant code, e.g., using stored procedures with typed parameters.

Numeric Precision: Floating point numbers do not exactly represent real numbers, and the imprecision between complex computations over the reals and over floats can become significant. For example, the failure of the Patriot missile system (resulting in the deaths of 28 American soldiers) was due to cumulative imprecision in a floating point timing routine [12]. Determining the loss of precision is therefore a common analysis that must be performed, usually manually, over embedded systems code.

Using symbolic execution, it is possible to describe imprecision at a per-path level. A näive approach would take the symbolic path and concretize it and then use interval analysis to examine the imprecision. It is not enough to enumerate the paths, however: one must examine the range of concrete values that are possible instantiations of the path in order to bound the precision errors that are possible. FITE should include a plug in that can generate both a constraint describing worst-case precision errors and a test (or series of tests) that demonstrate imprecise paths.

Concurrency: It is difficult to reason compositionally about concurrency using most programming languages. However, concurrency bugs are among the most expensive to detect and fix. Recent work, such as Symbolic Deadlock Analysis [8], can describe contacts between libraries and clients that guarantee deadlock free execution. This work is scalable enough to analyze large systems (1M SLOC Java / hour). Once the contracts are known, it is possible to cheaply analyze violations of the contract and present the result as a test case.

However, the other main concurrency problem of data races currently has no simple analysis solution. A task that could execute in the background and do pairwise analysis of "likely" concurrent method calls that could involve data races using a tool like CHESS [3] could be extremely valuable. The research challenges here involve the scale of the analysis and determination of "likely" interacting methods.

3.3 Regarding the user interface

Static analysis can determine potential program errors. Today, a typical IDE shows error messages of the type checker and other static analysis tools, relating them to particular lines in the code. We propose to augment this information with information gathered from and related to test cases. This includes already existing test cases as well as new test cases generated in addition. The additional information is meant to guide the developer towards errors directly related to the code the developer is currently working on. It is important that the additional information does not distract the developer from the main objective of writing code. Only relevant information must be shown. If a test case exposes an error, the code editor will associate the corresponding line in the code with the failing test case, also showing the stack trace of the failure. In addition to existing test cases, FITE generates new test cases, e.g. with (dynamic) symbolic execution. New test cases can be generated from scratch, or by "fuzzing" existing tests. When generating new tests, we will leverage the semi-automated analysis of interface boundaries and contracts. The analysis of existing test cases, and the generation of new test cases may happen continuously in the background, possibly on spare cores of modern multi-core machines, or the analysis may be delegated to the cloud. The developer can choose to include generated test cases into a regression test suite with a single button click, as for example shown in Figure 1.

Since environment abstraction, i.e. automated generation of mock objects, may cause generation of test cases with spurious errors, i.e. errors which cannot occur in the integrated system, we propose a ranking of generated tests, and their failures. The result can be visualized by a "heat map" in the editor, which illustrates the points in the code at which generated tests cause failures, showing those failures which are most likely to reproduce in the integrated system in the most threatening color. When test cases cause a failure, an automated failure root cause analysis determines the failure condition, and suggests to the developer the addition of a precondition into his code, effectively raising the failure to the abstraction level of the code the developer is currently working on. This has already been realized in Pex, as shown in Figure 2.

When the developer write new test cases, the editor will give suggestions what methods to call in a new test case. To this end, existing test cases and their code coverage are analyzed in the background to determine which methods of the product code or underrepresented in the existing test cases.

4. PROCESS ISSUES

So far the discussion has been mainly focused on unit analysis, from the developer's point of view. However, the ap-

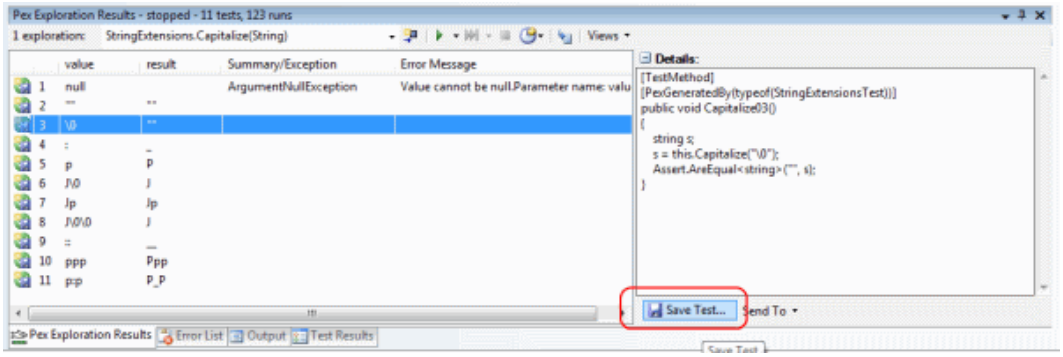

Figure 1: Saving an automatically generated test input.

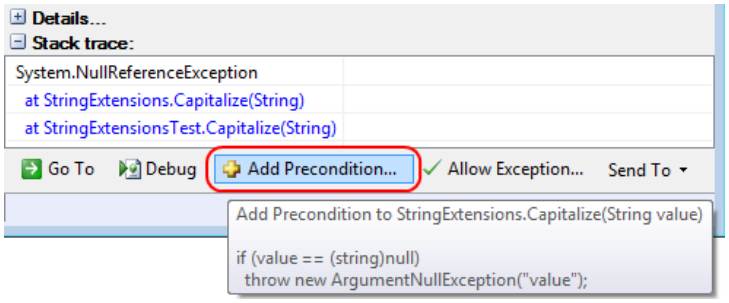

Figure 2: Adding an automatically generated precondition to a method

proach and the environment we envision should assist developers and testers during the whole application development process. It would be particularly effective to anticipate possible integration issues at the time of developing each single module composing the entire system.

The FITE environment will base its analysis strategies and corresponding results taking into account also the other modules to which the module under development is interrelated. To do this the FITE environment will need to be implemented as a distributed and collaborative environment running on the cloud. In this phase particularly important are possible interactions with legacy modules to be integrated within the system. For such modules FITE will need to include an analysis step to investigate and highlight possible integration issues. The analysis is performed on-the-fly while the developer is coding the module, analyzing the consequences of the decisions on legacy module usage.

The analysis can successively be pushed even further permitting to derive integration test cases covering possible interaction sequences within the system when component are ready to be released. The integration steps will be supported by FITE also through the semi-automatic derivation of stubs for the different test cases.

4.1 From unit to integration / system testing

While a unit test targets a single isolated features, a system test spans multiple features. Via semi-automatically inferred interface boundaries/contracts, FITE is able to assist the developer locally with unit tests, where those parts of the system not currently under test are mocked. In addition to traditional interface contracts, we propose to augment interface descriptions with a facility that allows to turn such mock instances into real instances. For example, when the database was mocked while testing a web application, then it should be possible to turn such a mock database state into an actual database state. This will in effect allow to turn unit tests into integration tests.

5. CONCLUSION

Current approaches for test generation and static analysis are used at the back end of the software development cycle, which makes them difficult to scale and difficult to use. In this paper, we propose an alternate vision of a V & V process that is performed continuously as the software is created. To make these ideas concrete, we introduced the notion of FITE, a future integrated test environment. This notion helps to crystalize several fruitful directions of research. We believe that such an environment, if available, would spur the widespread adoption of rigorous static and exhaustive analysis techniques, and significantly reduce costs and cycle times for developing high quality software in the future.

Acknowledgements: This paper was based on a whitepaper produced for the March 2010 *Practical Aspects of Software Testing* Dagstuhl Seminar. Many thanks to Mark Harmon, Henry Muccini, Wolfram Schulte, and Tao Xie for organizing an excellent seminar.

6. REFERENCES

[1] S. Anand, C. Pasareanu, and W. Visser. JPF-SE: A symbolic execution extension to Java PathFinder. In *Proc. of 13th TACAS Conf.*, volume 4424, page 134. Springer, 2007.

[2] N. Ayewah and W. Pugh. The google findbugs fixit. In *ISSTA '10: Proceedings of the 19th international*

symposium on Software testing and analysis, pages 241–252, New York, NY, USA, 2010. ACM.

[3] T. Ball, S. Burckhardt, K. Coons, M. Musuvathi, and S. Qadeer. Preemption sealing for efficient concurrency testing. In *Proceedings of the 16th Annual Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, March 2010.

[4] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. R. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[5] B. Boehm. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[6] A. Chakrabarti and P. Godefroid. Software Partitioning for Effective Automated Unit Testing. In *Proceedings of EMSOFT'2006 (6th Annual ACM & IEEE Annual Conference on Embedded Software)*, pages 262–271, Seoul, October 2006. ACM Press.

[7] S. Cherem and R. Rugina. A practical escape and effect analysis for building lightweight method summaries. In *In CC 2007: 16th International Conference on Compiler Construction*, pages 172–186, 2007.

[8] J. Deshmukh, E. A. Emerson, and S. Sankaranarayanan. Symbolic deadlock analysis in concurrent libraries and their clients. In *ASE '09: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 480–491, Washington, DC, USA, 2009. IEEE Computer Society.

[9] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Proceedings of NDSS'2008 (Network and Distributed Systems Security)*, pages 151–166, February 2008.

[10] P. Godefroid, A. Nori, S. Rajamani, and S. Tetali. Compositional May-Must Program Analysis: Unleashing The Power of Alternation. In *Proceedings of POPL'2010 (37th ACM Symposium on Principles of Programming Languages)*, Madrid, January 2010.

[11] S. P. Miller, M. W. Whalen, and D. D. Cofer. Software model checking takes off. *Commun. ACM*, 53(2):58–64, 2010.

[12] T. U. S. G. A. Office. Patriot missile defense report. Technical Report, B-247094. Available at: http://www.fas.org/spp/starwars/gao/im92026.htm, February 1992.

[13] D. Saff and M. D. Ernst. Reducing wasted development time via continuous testing. In *proceedings of the 14th International Symposium on Software Reliability Engineering*. IEEE Computer Society, 2003.

[14] N. Tillmann and J. de Halleux. White-box testing of behavioral web service contracts with pex. In *TAV-WEB '08: Proceedings of the 2008 workshop on Testing, analysis, and verification of web services and applications*, pages 47–48, New York, NY, USA, 2008. ACM.