OnSpot System: Test Impact Visibility during Code Edits in Real Software

Muhammad Umar Janjua
Operating System Group R&D, Microsoft Corporation, Redmond, US
mujanjua@microsoft.com

***Abstract*— For maintaining the quality of software updates to complex software products (e.g. Windows 7 OS), an extensive, broad level regression testing is conducted whenever releasing new code fixes or updates. Despite the huge cost and investment in the test infrastructure to execute these massive tests, the developer of the code fix has to wait for the regression test failures to be reported after checkin. These regression tests typically run way later from the code editing stage and consequently the developer has no test impact visibility while introducing the code changes at compile time or before checkin. We argue that it is valuable and practically feasible to tailor the entire development/testing process to provide valuable and actionable test feedback at the development/compilation stage as well. With this goal, this paper explores a system model that provides a near real-time test feedback based on regression tests while the code change is under development or as soon as it becomes compilable. OnSpot system dynamically overlays the results of tests on relevant source code lines in the development environment; thereby highlighting test failures akin to syntax failures enabling quicker correction and re-run at compile time rather than late when the damage is already done.**

We evaluate OnSpot system with the security fixes in Windows 7 while considering various factors like test feedback time, coverage ratio. We found out that on average nearly 40% of the automated Windows 7 regression test collateral could run under 30 seconds providing same level of coverage; thereby making OnSpot approach practically feasible and manageable during compile time.

Categories and Subject Descriptors—D.2.5[**Software Engineering**]: Testing and Debugging

General Terms—Reliability, Design, Experimentation

Keywords—software; testing; analysis; code writing; development; early regression; real product testing

I. INTRODUCTION

The fact is that the bugs are mostly added to the source code at the moment it is fixed or changed. The introduction of a change in the source code is crucial and risky. We believe that the right time to fix these bugs should ideally be closed to the time code change is added. Large enterprise organizations like Microsoft employ extensive, broad level of testing [7] to maintain the quality of the updates to existing products like Windows 7. Such organizations invest lot of time and resources in the test infrastructure, execution and test pass completion so that all the change related and component wide scenarios as well as dependent systems and diverse third party applications could be covered and confirmed to be regression free. While there is a lot of useful information available about successful/failing tests, coverage percentage for the changed lines etc, it is reported only after the developer has completed coding and made the checkin. Instead, we see a potential value and use of test related feedback at compile time as well and argue to tailor the entire testing process to provide test related feedback at the development/compilation stage as well. The advantage of providing such an on-spot feedback to the developer is to detect and prevent runtime failures and regressions nearly at the same time as the code edit happens. Otherwise, such failures are normally discovered at the code check-in time or by full test passes that incur higher costs. By that time, the damage is already done.

With OnSpot system, we provide a unified, seamless and lightweight interface to developer that pulls relevant tests from diverse components and dependent software and provides precise test feedback about the current change being introduced at the compile/development time. This approach makes distant regression testing an integral part of the change introduction process, or a first class citizen for developer, where testing would not be taken as a post-checkin activity, but as a mechanism to seek insight about the code change right on the spot, and before it is checked in or shared.

If we compare with local unit testing available to developers, although these can complete faster yet these are limited to the component scope only and the included tests lack system wide breadth and diversity. So it also becomes critical to conduct relevant regression testing early for scenarios that are not covered with simple unit tests. Many a times local unit testing succeeds but the buddy software tests fail way later. With OnSpot system, we bring the test diversity and greater coverage from large scale system wide regression tests and make it available to the developer at code editing stage in a lightweight and actionable manner.

The contributions of this paper are:

1. First to propose and investigate the need for a lightweight model to run regression tests at compile time while introducing code changes in the light of test failures.
2. Implements the proposed model as a working prototype system "OnSpot" based on the real world Windows 7 test system that establishes a near real-time link between the source changes and the state of regression test execution in one unified interface.
3. Evaluates the practical feasibility of the approach by running several real world security fixes shipped in Windows 7.

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2804430

OnSpot system is compatible with any existing test selection/prioritization/minimization techniques and can integrate well with traditional software development processes with a little effort.

Table 1: Comparing characteristics of OnSpot with other

For complex products(millions of line of code)	**OnSpot**	**Syntax/ Type Checker**	**Prefast/ Static Code Analyzer**	**Local Unit testing**	**Traditional Testing**
Lightweight	**Y**	Y	Y	Y	N
Bug Prevention Stage	**Code editing**	Code editing	Code editing	At checkin	After Check-in
Feedback time	**Immediate (somewhat)**	Immediate	Immediate		Delayed
Change sensitive	**Y**	Y	Y	Y	N
Precision	**Y**	Y	Approximate	Y	Y
Corrective Action at dev box	**Y**	Y	Y	Y	N
Test Diversity	**Across entire system**	n/a	Limited static checks	Only component	Across **entire system**

II. RELATED TOOLS AND TECHNIQUES

Currently, the type of static tools available to developers that can be used to assess the effect of a code change are quite limited in their feedback. They only address limited classes of bugs and properties about the programs during development. For example, compiler techniques, like syntax, semantic/type checking can only determine error typos, missing variable, undefined functions, incorrect casts etc. Static source code analyzer like Prefast [6] take a step higher and can provide approximate feedback for example about buffer overflows and pointer allocation errors. But, what remains elusive from the prior techniques are more interesting use cases and richer class of program properties and behaviors, captured as test cases. Currently at static time, the developer has no clue how its current code change affects the relevant test cases.

On the other end, there are series of profile driven tools relying on code coverage information[5] that can link up source code change with the related tests. However, these tools operate at post-build time, are applicable after the fix has been checked in, and currently do not help the developer directly during code changes. Further, there has been extensive work and techniques [2,3,4] on selecting, minimizing and prioritizing tests based on code coverage. OnSpot system does not modify or build new techniques in this category. However, it is compatible with any of these techniques, which can be incorporated in the OnSpot system.

III. MODEL FOR CODE CHANGE INTRODUCTION PROCESS

We define a number of important characteristics for OnSpot system and the model for change introduction. Refer to Table 1, and Figure 1, respectively.

```
$for\ i = 1 \dots f$
        $binary_{c_i} = incrBuild(c_i)$
        $T_{c_i} = selectTests(\ Ccov(binary), binary_{c_i})$
        $Res_{c_i} =\ RunTests(T_{c_i});$
        $ShowTestFeedback(\ Res_{c_i});$
```

Figure 1: Model for change introduction

Consider the introduction of fix as a series of code changes with interleaved compilations. Let $\cup c_i$ be the set of changes where $c_i = code\ change\ at\ i^{th} compile\ time$ and $c_i R\ c_{i+1}$ such that $R \epsilon \{\subseteq, ! \subseteq\}$. The relation R could be taken as a syntactic subset. Let c_f be the final fix which is supposed be checked in to the source depot.

A. Change granularity (Testable change)

A change becomes a candidate for test when it is compilable. The developer however can control the point when he wants invoke the test during his fix. This might vary with his thought process, and current understanding of his partial fix so far.

B. Automatic Build and deploy

We propose an incremental build process *incrBuild(* c_i *)*,that produces a modified binary $binary_{c_i}$, containing the code change c_i.

C. Test Selection and Coverage ratio

Assume we have a original code coverage data for the *binary* before any change c_i is introduced . Let $\boldsymbol{T}$ be the set of all tests available for this binary. We define a procedure *selectTests(CCov(binary),* $binary_{c_i}$ *)*, which takes the modified binary and the code coverage data, and determines the tests $T_{c_i} \sqsubset \boldsymbol{T}$ that are relevant to the change c_i . We define code coverage ratio to be the number of binary blocks covered by tests divided by the total number of blocks added or modified as a result of introducing change. In practice, OnSpot uses Echelon's [13] test selection method that analyzes the difference between the old and new binary in the light of previously stored baseline of code coverage data. In the first step, Echelon identifies impacted binary blocks(new plus old modified) using binary level differencing [14] between the two versions. Then prior code coverage information is used to select tests that cover the old modified block directly or cover the old immediate predecessor/successors of the newly added block in the new binary.

D. Test Setup and Execution

Once the related tests are identified, the tests T_{c_i} are automatically setup and run and the results/logs are collected. Let Res_{c_i} be the test result set for executing tests.

E. Feed Back

The developer has a pipeline of Res_{c_i} at his end for the series of changes c_i introduced so far. At any i-th stage, the developer is aware how the current change impacts the selected test results.

Table 2: Result pipeline for developer

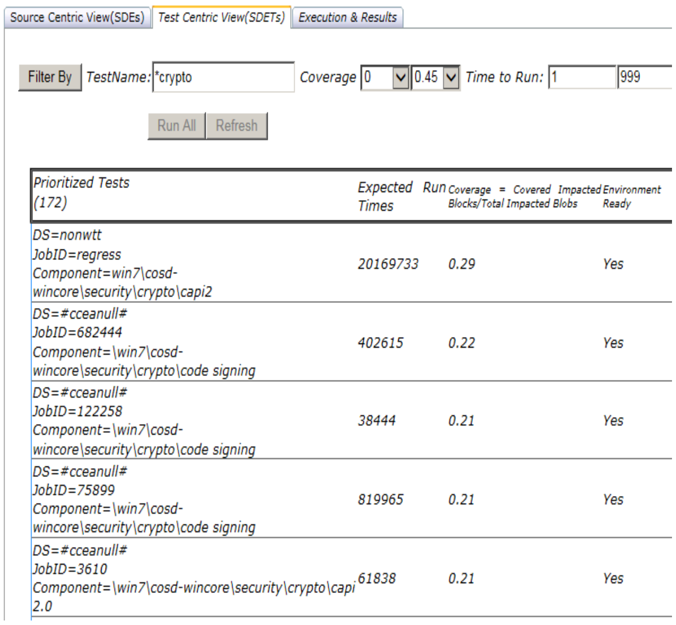
Code change	Relevant Tests	Pass %	Results
c_i	T_{c_i}	90%	Res_{c_i}
c_{i+1}	$T_{c_{i+1}}$	80%	$Res_{c_{i+1}}$
….	….	….	….
c_f	T_{c_f}	100%	Res_{cf}

F. Test Response Time

For change c_i , let the response time RT(c_i) be the total time it takes run tests and show test feedback to the developer. The goal is to show relevant tests with low RT (c_i) and better coverage for developer for execution.

IV. Architecture and Implementation

In this section, we layout the system components which implement the above model. First, a workflow is established on the developer's box that monitors code changes and compiles these automatically. The changed binaries are passed to the backend where it is analyzed



TestName: *crypto Coverage 0 0.45 Time to Run: 1 999

Prioritized Tests (172)	Expected Run Times	Coverage = Covered Impacted Blocks/Total Impacted Blobs	Environment Ready
DS=nonwtt JobID=regress Component=win7\cosd-wincore\security\crypto\capi2	20169733	0.29	Yes
DS=#cceanull# JobID=682444 Component=\win7\cosd-wincore\security\crypto\code signing	402615	0.22	Yes
DS=#cceanull# JobID=122258 Component=\win7\cosd-wincore\security\crypto\code signing	38444	0.21	Yes
DS=#cceanull# JobID=75899 Component=\win7\cosd-wincore\security\crypto\code signing	819965	0.21	Yes
DS=#cceanull# JobID=3610 Component=\win7\cosd-wincore\security\crypto\capi 2.0	61838	0.21	Yes

Figure 2: Test centric view

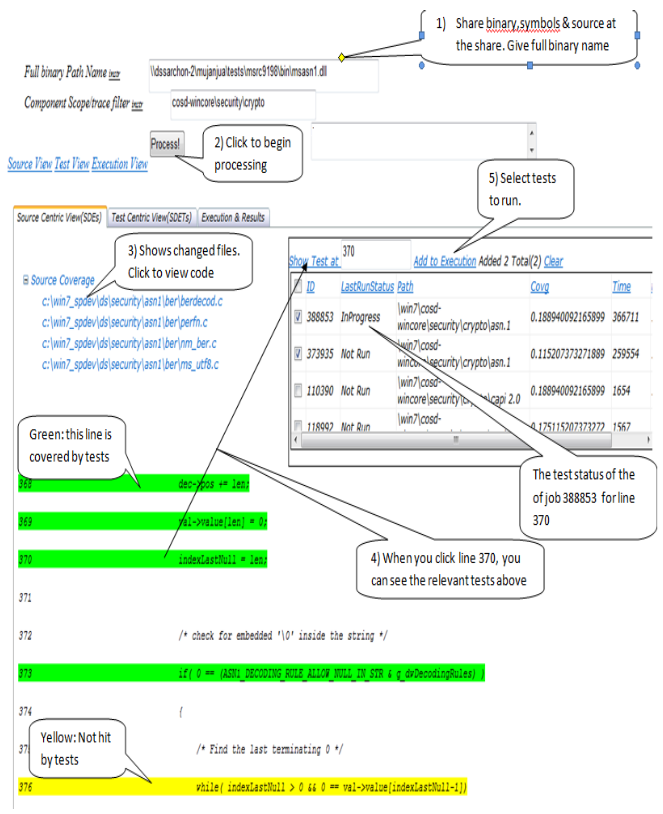
for changed binary blocks and determination of related tests that cover that change. These tests with the associated coverage ratio, execution time, setup complexity, component scope are shown to the developer. The developer has two main views.

A. *Test Centric View*

In this view, the developer has a complete view of all the regression tests that cover the code change being introduced. These tests are selected and prioritized according to the level of automation, expected execution time, coverage ratio and environment readiness. The developer has a filter to select and search from the tests.

B. *Source Centric View*

The source centric view in Figure 3 provides an overlaying of the relevant tests and their results over each changed source line. This enables developer not only to see how its current code changes introduced are affecting the selection of tests, but also how results of test runs change (from fail to pass or vice versa) as he progresses towards the completion his fix. This brings a near instant link between test results and code editing. The regression test result become a kind of first class citizen in the developer's IDE. The source lines marked green have associated tests that cover these lines. The lines marked yellow indicate no existing regression tests cover them. This helps developer figure out how to better write future tests that can cover yellow lines to improve test coverage.


1) Share binary,symbols & source at the share. Give full binary name
2) Click to begin processing
3) Shows changed files. Click to view code
4) When you click line 370, you can see the relevant tests above
5) Select tests to run.
The test status of the of job 388853 for line 370
Green: this line is covered by tests
Yellow: Not hit by tests
Process!
Source View Test View Execution View
Source Centric View(SDEs)
Test Centric View(SDETs)
Execution & Results
Source Coverage

Full binary Path Name: \\dssarchon-2\mujanjua\tests\msrc9198\bin\msasn1.dll

Component Scope/trace filter: cosd-wincore\security\crypto

c:\win7_spdev\ds\security\asn1\ber\berdecod.c
c:\win7_spdev\ds\security\asn1\ber\perfn.c
c:\win7_spdev\ds\security\asn1\ber\nm_ber.c
c:\win7_spdev\ds\security\asn1\ber\ms_utf8.c

Show Test at 370 Add to Execution Added 2 Total(2) Clear

	ID	LastRunStatus	Path	Covg	Time
✓	388853	InProgress	\win7\cosd-wincore\security\crypto\asn.1	0.188940092165899	366711
✓	373935	Not Run	\win7\cosd-wincore\security\crypto\asn.1	0.115207373271889	259554
	110390	Not Run	\win7\cosd-wincore\security\crypto\capi 2.0	0.188940092165899	1654
	118992	Not Run	\win7\cosd-	0.175115207373272	1567

```
368                 dec->pos += len;
369                 val->value[len] = 0;
370                 indexLastNull = len;
371
372                 /* check for embedded '\0' inside the string */
373                 if( 0 == (ASN1_DECODING_RULE_ALLOW_NULL_IN_STR & g_dwDecodingRules) )
374                 {
375                     /* Find the last terminating 0 */
376                     while( indexLastNull > 0 && 0 == val->value[indexLastNull-1])
```

Figure 3: Source Centric view with overlayed test results

V. Case Study: Analysis and Discussion

We have evaluated several Window 7 code fixes and successfully run these through the OnSpot system to figure out its practical usefulness and applicability. For illustration purpose and demonstrate the developer's experience, we shall take one representative Windows 7 security fix [1]. This security fix modifies the string decoding function for ASN.1 standard implementation in the windows (msasn1.dll). The fix detects for the null in the string encoding and handles it appropriately during parsing in lower level ASN decode string functions. This is a typical security fix and on average the numbers calculated should be valid for other fixes in the same security category with very little variation. We used Amd dual core 8GB RAM machine as a developer box that was connected to coverage/build/test systems.

To evaluate the various characteristics of the OnSpot model proposed earlier, we set four main evaluation criteria..

A. *What was the average time to identify relevant tests during code edits and how it varies with increasing component scope in OnSpot?*

As the developer compiles the fix, OnSpot starts building up the test selection for the given change. From the entire window 7 OS regression tests, OnSpot selected 11,484 test jobs that covered the modified binary blocks of the fix in the new msasn1.dll. Each of these jobs could typically contain hundreds of test cases itself.

Table 3: Time for identifying relevant tests in OnSpot

Scope of Tests	**Avg. Time(s) for SelectTest()**	**#of T_{c_i} (relevant tests)**
WinCore	<30	11484
--Security	<10	871
-----Crypto	<9	172
-------- MSASN1	<8	13

As shown in Table 3, the time taken for SelectTest increased expectedly as the test scope increased from smaller component msasn1.dll to the entire windows core, but the interesting aspect to note was that the number of T_{c_i} increased in multiples of tens in comparison to just a second increase in allowed time for SelectTests. It was critical to show developer how his compilable code changes impact the test selection. Our experience with OnSpot demonstrated that large number of relevant regression tests that were not visible before at compile time were selected and shown in a light weight and quicker fashion with lower idle time for developer (from 8 sec).

B. *How diverse the test selection became across the system?*

The relevant tests selected belonged to diverse set of components across windows core like kernel, networking, storage, file systems etc. This provided an opportunity for substantially broader coverage across diverse components at development time than possible with unit/regression tests of one component (MSASN1) alone. In this fix, test selection came from a scope of around 800 window components.

C. *How the test covergage increased with OnSpot?*

Table 4: Tests and scope contributing to coverage

Covg_ratio	%of T_{c_i}
0.00-0.05	0.83%
0.05-0.10	**54.96%**
0.10-0.15	**13.29%**
0.15-0.20	**25.91%**
0.20-0.25	5.01%
0.25-0.30	0.01%

Scope	Max Cvg
WinCore	0.30
--Security	0.29
-----Crypto	0.29
--------MSASN1	0.18

The developer can then look at the coverage provided by the test jobs, and also at the number of tests that can run. He can view the tests relative to the source line and then runs and selects these tests for running and retrieving test results.

Nearly 95% of the relevant tests had at least 5% of the coverage for the fix. Further, at most 30% of the code changes in this fix had some sort of regression tests covering for it which was better than 18% coverage alone with msasn1 tests only. Interestingly, as the component scope for the tests increases, we see that the coverage ratio increases as well. This clearly demonstrated the advantage with OnSpot system that through these additional regression tests it provided 12% more coverage at compile time that would not have been possible with component specific tests only.

D. *What percentage of the tests could run in reasonable time during code editing ?*

One of the critical requirements for the OnSpot system to be practically useful was to have large proportion of selected tests that could complete the execution run in reasonable time as the code changes became compilable. As per Figure 4, it turned out to be an encouraging percentage and clearly demonstrated that nearly half of the regression tests could provide quicker feedback during code edits intervals of 30 seconds or more and nearly a quarter of the tests for code edits intervals of 15 seconds. Furthermore, we learnt that by running these tests, which complete execution within 30 seconds it was still possible to achieve fix coverage of 25 percent that was closer to the maximum coverage of 30 percent if all the regression tests would have run. The choice of the right size of code editing interval depends on how the developer writes the code and if there are regression tests that can complete within that time. Since we have evaluated thousands of Windows 7 core OS regression tests, we are confident that for regression test suites of such scale, there will always be substantial number of tests that can complete within the preferred code editing intervals.

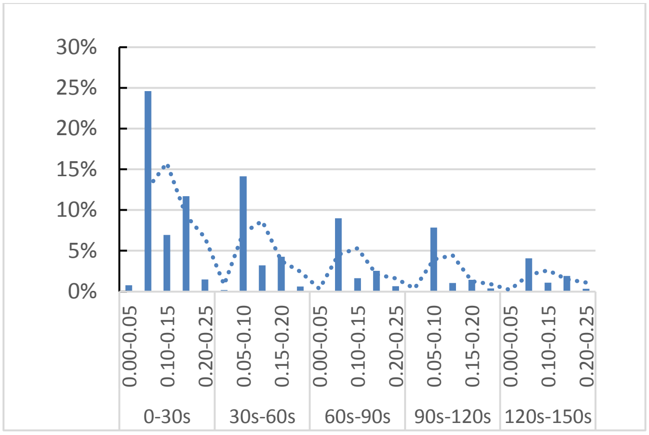


Figure 4: Percentage of tests completing under varying code editing intervals with achieved coverage ratio

VI. Conclusion

We have provided a simple lightweight model to bring distant regression testing at compile time to help developer detect and prevent bugs at the moment the change is introduced in a real software. By overlaying test results, coverage ratio over the changed source lines, the developer has better comprehension of the effect of the code changes on the test results and wider selection of tests to run to achieve higher coverage than possible with local unit or component only tests We believe that organizations with complex and huge test collateral should employ OnSpot system to detect possible test failures early to avoid later costs. Our experience with OnSpot shows that it's practically manageable and helpful for developers to provide regression test feedback during reasonable code editing intervals.

References

[1] http://blogs.technet.com/b/srd/archive/2009/10/12/ms09-056-addressing-the-x-509-cryptoapi-asn-1-security-vulnerabilities.aspx

[2] S. Yoo and M. Harman, "Regression testing minimization,selection and prioritization: A survey," Software Testing, Ver-ification and Reliability, vol. 22, no. 2, pp. 67–120, 2012.

[3] G. Rothermel and M. J. Harrold, "Empirical studies of a safe regression test selection technique," IEEE Transactions on Software Engineering, vol. 24, no. 6, pp. 401–419, Jun. 1998.

[4] J.-M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test application frequency," in Proceedings of the 22nd c conference on Software engineering,2000, pp. 126–135.

[5] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal, "A study of effective regression testing in practice," in Proceedings of the Ei1ghth International Symposium on Software Reliability Engineering, 1997, pp. 264–274.

[6] http://research.microsoft.com/enus/news/features/prefast.aspx

[7] Jacek Czerwonka , Rajiv Das , Nachiappan Nagappan , Alex Tarvo , Alex Teterev, CRANE: Failure Prediction, Change Analysis and Test Prioritization in Practice -- Experiences from Windows, Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation, p.357-366, March 21-25, 2011

[8] A. Srivastava and J. Thiagarajan, "Effectively Prioritizing Tests in Development Environment," *Proc. ACM SIGSOFT Int',l Symp. Software Testing and Analysis (ISSTA ',02),* pp. 97-106, 2002