Introducing Continuous Selective Testing of Evolving Software

Version 0.2

Mary Jean Harrold, Darko Marinov, Stephen Oney, Mauro Pezze, Adam Porter, John Penix, Per Runeson, Shin Yoo

Introduction

Current trends in software engineering, towards more and more continuous evolution of software, using agile methods, rather than distinct development phases, put new requirements on software testing. When new versions of the software are built every day, hour or minute, the test process must change accordingly. The time when regression test suites were run at the end of the project is passed. Test-driven development and continuous integration practices are designed to meet those needs in agile development. Test automation is a key in introducing these practices and is often a key desire in test improvement (Runeson, 2006).

These testing practices provide a general approach to meet the needs, but even the automated tests will sooner or later be too time consuming to be run completely every time, and some kind of test selection must take place. When the regression test suite takes longer than a weekend, the build test suite takes more than overnight, or when the unit test suite takes more than a coffee break, there is a need for selecting subsets of test cases in a systematic way, to enable efficient software engineering.

Other trends also drive the need for efficient test selection, namely highly configurable systems and product line architectures, which offer an enormous variability that causes the testing space to grow. Similarly, globally distributed development adds cultural and other communications barriers to the complexity of the complexity of the software. Communication of changes and dependencies in the software can be achieved by continuous testing, complemented by feasible communication tools.

The Context

After modifying software, developers typically want to know that the changes have the desired effect. Developers often do some form of incremental testing to ensure this. The simplest incremental testing strategy is to rerun all existing test cases. This method is simple to implement, but may be unnecessarily expensive, especially when changes affect only a small part of the system. Consequently, many regression test selection (RTS) techniques have been proposed (Yoo and Harman, 2010, Engström et al. 2010). With these approaches only a subset of test cases are selected and rerun. Since, in general, optimal test selection (i.e., selecting exactly the fault-revealing test cases) is impossible, the cost-benefit tradeoffs of RTS techniques are a central concern of incremental testing in practice.

Our understanding of these techniques, however, is limited. One reason is that incremental testing, by definition, is a recurring process, not a one-time activity. As a result, the costs and benefits of a given technique can change over time. Another reason is that project-specific process and product characteristics can greatly affect costs and benefits. That is, the fit of a given testing technique will vary across projects.

Dagstuhl Seminar Proceedings 10111
Practical Software Testing : Tool Automation and Human Factors
http://drops.dagstuhl.de/opus/volltexte/2010/2622

Below we describe some factors that affect the costs and benefits of using incremental testing techniques.

Process and Product Characteristics. Individual projects can vary in many ways. For example, projects following an agile process will divide the project into many, short increments. Those following a more waterfall-like process will have few, longer increments. Within an increment, some projects will have high code churn rates, with a high percentage of code changing, producing new *versions* of methods, files and systems at a very high pace. Others may be more stable, changing only a small percentage of the code in any given increment. Some processes will use co-located developers, for whom communication is easy to organize. Other projects will use distributed development teams, relying more heavily on the results of incremental testing to signal and communicate development status and activity. Software architectures can vary. Some projects will be created from many, independent, but statically known components; some will be created from independent components that are dynamically composed at runtime; other projects will have more monolithic architectures; and others will be highly configurable. In any case, many *variants* may exist simultaneously with smaller or larger differences between them. Finally, different projects will support different non-functional requirements, such as being safety critical, needing high availability, or requiring an ability to operate on multiple different computing platforms.

Testing Goals. The purpose of each individual testing session can vary as well. Some testing activities will focus on testing functional correctness, while others will focus on non-functional requirements. Functional testing is often divided into different levels of testing granularity. Some test acticites are focused on individual code units; some on testing integrated assemblies of components; some on testing the whole system. Non-functional testing may look at issues such as performance and operation under heavy loads.

Testing Resources. Test suites often start small, but build up over time. Since developers generally want feedback as quickly as possible, they sometimes manually subdivide their test suites into smaller test suites. For instance the smallest-sized test suites are sometimes called smoke tests. Smoke tests are few and of short duration, taking say 5 to 15 minutes to complete. The next sized test suites may run in 1 to 4 hours. The next larger may be intended to run over night, taking up to 8 hours. Finally, the largest test suites can take many days over even months to complete.

Testing Tools. The tools referred to when talking about testing tools are often test *execution* tools and frameworks, i.e. the jUnit framework. However, in an iterative development environment, configuration management tools are key to keep track of versions and variants of software to be tested. Further, test management tools, keeping track of the planning and execution of test cases is needed to support decision and monitoring of the test process and what is tested, by whom, and when. Defect management tools also play an important role in following up test activities.

General Infrastructure

When introducing continuous testing of evolving software, which is selective, based on systematic approaches, the testing infrastructure must meet some specific requirements to enable the testing. We define a general infrastructure for iterative testing, with process and tools.

Test process: The iterative testing process begins with two versions of the source code and test code, see Figure 1. Version N is the most recent version of the code that has been tested. Version N' is a newer version of the code that has undergone one or more changes. Both versions are provided as input to an Impact Analysis activity that determines which parts of the code are potentially affected by the change. The output impact analysis is some collection of artifacts from the code – such as source lines, methods, or files – that are considered impacted by the change, i.e. constitute Affected Artifacts. Next, a Test Selection activity takes the Affected Artifacts and determines a subset of the test to be executed, based on the information available on the change, test execution history etc. Then, a Test Prioritization activity orders the test, based on similar information. Then the Ordered Set of test cases are executed, either fully, or as long as time permits. The result of Test Execution includes test result (pass, fail, timeout, etc.) as well as other information such as code coverage data and test run duration data. These data are used for the next iteration of testing.

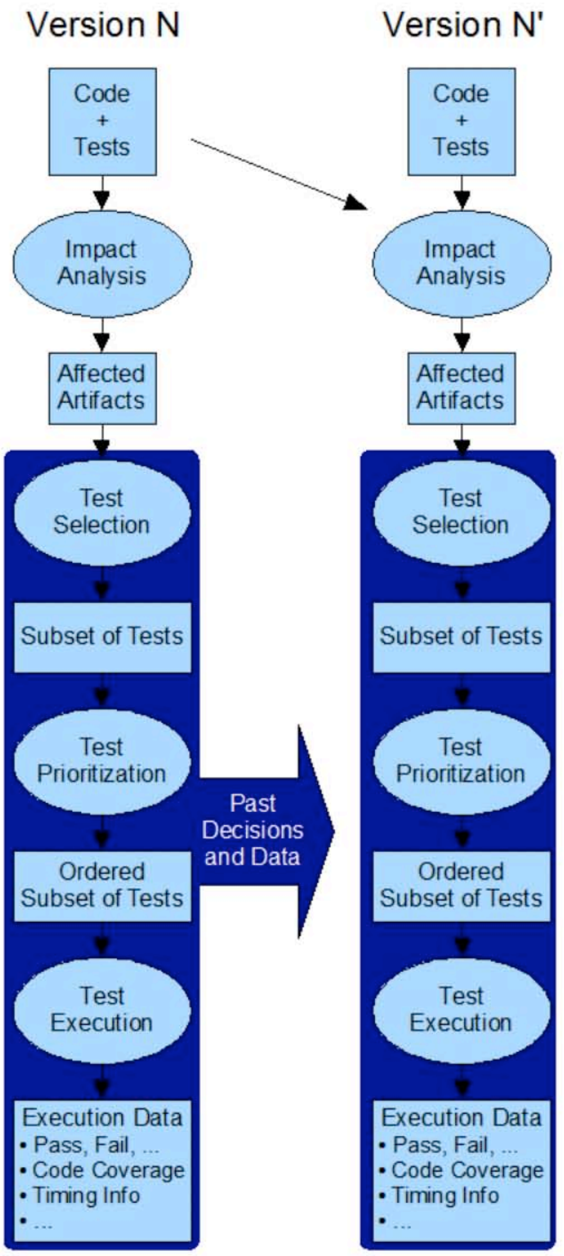

Figure 1. Conceptual process for continuous selective testing.

Tools: In order to support this process, the tool chain must be adapted. The tools needed are of three kinds, configuration management (CM), defect management (DM) and test case management (TCM), and specific relations between them must be in place, see Figure 2.

Configuration management tools must support the definition of versions at different levels – method, file, component or system – depending on the context. The CM tools must also support the definition of product variants. Build dependency relations must be supported by the CM tools to enable identification of underlying components in the impact analysis. Finally, it must support the identification of changes between versions of configuration items, to produce a list of artifacts affected by the changes between two versions. Test cases must be set under version control, like any software engineering artifact.

Defect management tools must store information about defects found and corrected, including information about which version and variant the defect was found. Defect types may also be stored to enable use of approaches that give different priority to different types of defects.

Test case management tools must store the execution history of test cases. This includes connections to CM tools to track which versions and variants are tested with a certain test case. Further, information about which test cases identified defects must be possible to track from each test case.

Based on this tool chain, a prioritized list of test cases can be derived based on their historical ability to detect faults, relation to recent changes or most recent execution time, depending on the prioritization strategy chosen. The test designer may use the proposed list of test cases as is, or adjust it based on other priorities in the project. For this purpose, the test selection tool must provide a scoreboard function to enable the testers' interaction with this list.

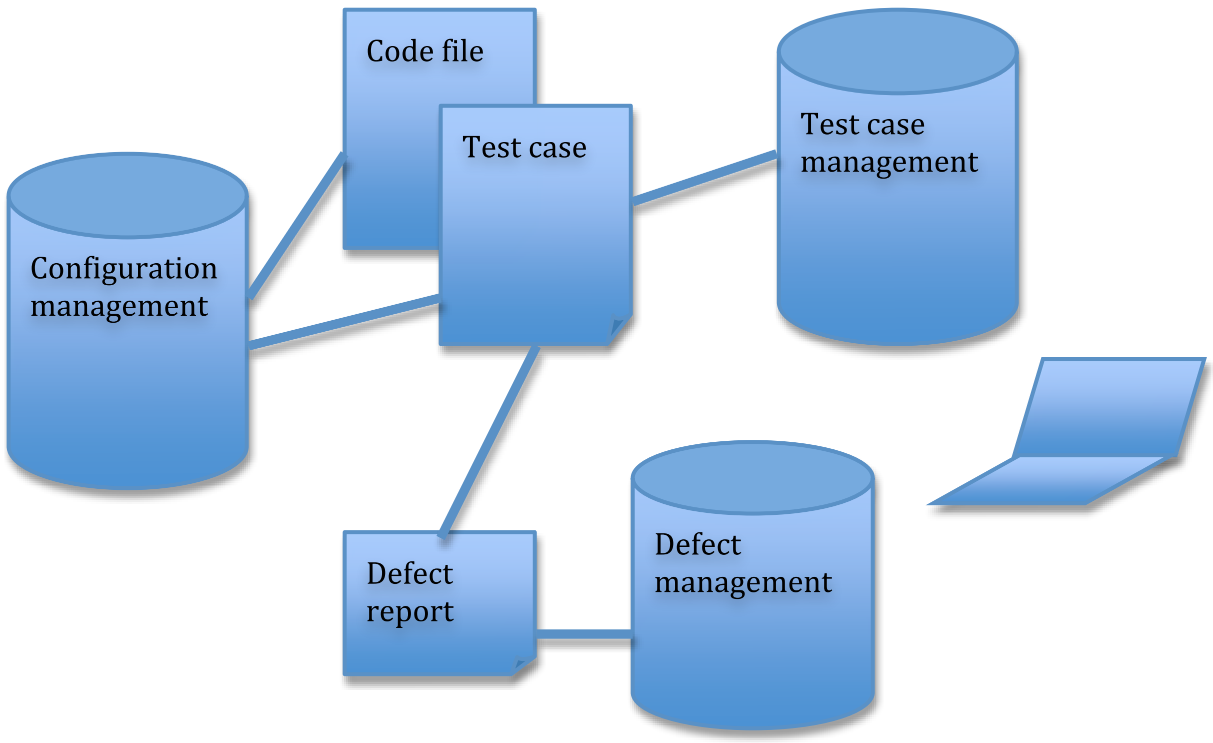

Figure 2. Tool support needed to support continuous selective testing.

Human factors. Human factors become an issue, especially when multiple developers are working on interdependent components. While coordination between developers normally happens formally, at predetermined intervals, which is time-consuming but predictable, or informally, which is unpredictable, but flexible. The idea of Continuous Coordination (http://en.scientificcommons.org/42468508) has been proposed to aid developers working on collaborative software engineering projects. The idea behind continuous coordination is that developers are given the freedom to coordinate themselves however they see fit, but that through information visualization, they can make better choices when synchronizing their codebases. One technique, for example, would allow developers to see what modules other developers are working on (information that is collected implicitly through developer activity). This would allow them to make more informed decisions if, for example, a test starts breaking, of whether or not and how they should integrate their code.

Case studies

Scenarios that show the applicability of the techniques
For each case, present (as much as secrecy policies allow):

- Goals – what will the company achieve by the continuous testing?
- Driving forces – what are the motivating factors? Who is driving?
- Solutions (related to general model)
  - At what levels is this applied (unit, system)
  - Which information is used for the selection and execution? Which tools?
  - Frequency of iterations? Magnitude of code and changes?
- Gains – what are the gains with the continuous testing? Is it worth the effort spent?

Selected cases

- Google (John Penix)
- ABB (Brian Robinson?)
- Microsoft (Wolfram Schulte, Nachi Nagappan?)
- ~~IBM (Amit Paradkar???)~~

1-2 pages per case?

Tools and Technologies – Research Review

Techniques for reducing the cost of testing evolving software can be categorized in two different ways: the methodology they use and the criteria they are based on. For more details, readers are encouraged to refer to recent surveys of the area (Engström et al. 2010, Yoo and Harman, 2010).

Methodologies for Reducing Cost of Testing

Selection Approach: Here, the goal is to reduce the number of tests to be executed by identifying the set of tests that are not relevant to the latest changes in the software. The relevance relation is determined based on various static analyses. For example, the control flow analysis is a criterion that is often used for unit level testing: the static analysis identifies the locations in Control Flow Graph (CFG) where the old version and the new version of the source code depart from each other. Test cases that execute these locations may potentially reveal the differences between two versions and, therefore, they need to be executed after the change. Empirical studies have shown that, if applied with the right frequency, selection approach can result in as much as XX% reduction of testing cost[cite].

Minimization Approach: The goal of minimization approach is to identify redundant tests. The redundancy of a test case is defined in relation to the set of test goals it achieves. A test case can be said to be redundant if there exist another test case that achieves all the test goals of the first test case and more. One metric that has been widely used as test goals is code coverage; if a test case covers a subset of structural elements (e.g. statements or branches) that are covered by another test case, the first test case is redundant in terms of code coverage.

Prioritization Approach: The goal of prioritization approach is to try to maximize the early fault detection so that the tester can get the maximum benefit even if the testing is terminated at an arbitrary point. Since the fault information is inherently absent at the time of testing, in practice the prioritization uses a surrogate property that is expected to correlate with the rate of fault detection. Again, one widely used surrogate is code coverage.

Criteria for the Cost Reduction

Code Coverage: Structural coverage of source code has been the most widely used metric that guides the aforementioned approaches. While the realization of code coverage is not sufficient to achieve fault detection, it is nevertheless necessary and, therefore, should be pursued as much as possible.

Historical Information: [description of usage of data such as the last test execution time, past fault coverage, etc in the literature]

Model-based Criteria: [description of selection/minimization/prioritization criteria defined over UML/FSM etc]

Conclusions

TBD

Acknowledgement

This paper originates from the Dagstuhl workshop 10111 Practical Software Testing.

References

E. Engström, P. Runeson and M. Skoglund, A Systematic Review on Regression Test Selection Techniques, *International Software Technology*, 52(1):14-30, 2010.
S. Kim, S. Park, J. Yun and Y. Lee, Automated Continuous Integration of Component-Based Software: an Industrial Experience, *23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 423-426, 2008
P. Runeson. A Survey of Unit Testing Practices. *IEEE Software*, 23(4):22, 2006.

S. Yoo and M. Harman, Regression Testing Minimisation, Selection and Prioritisation : A Survey, *Software Testing Verification and Reliability*, online March 2010, DOI: 10.1002/stvr.430