Continuous Selective Testing

Bastian Steinert, Michael Haupt, Robert Krahn, and Robert Hirschfeld

Software Architecture Group
Hasso Plattner Institute
University of Potsdam, Germany
firstname.lastname@hpi.uni-potsdam.de

Abstract. A manual and explicit activity, the frequent selection and execution of tests requires considerable discipline. Our approach automatically derives a subset of tests based on actual modifications to the code base at hand, then continuously executes them transparently in the background, and so supports developers in instantly assessing the effect of their coding activities with respect to the overall set of unit tests to be passed. We apply techniques of selective regression testing, mainly relying on dynamic analysis. By taking advantage of the internal program representation available in IDEs, we do not need to rely on expensive comparisons of different program versions to detect modified code entities.

1 Introduction

Test-driven development [5] (TDD) is a cornerstone of agile software development methodologies such as Extreme Programming [22] (XP). This technique suggests to write test cases before the code they are intended to cover. Written first, tests serve multiple purposes. First, they represent a specification for the system to be developed. Next, they document the system and help other developers in comprehending the system. Finally, they ensure that every single change violating one of the required features described in the executable form of a test is reported.

While testing is an important part of regular development activities, Integrated Development Environments (IDEs) have little support for selecting and (re)executing tests relevant with respect to modifications applied to the system under development [18].

There are a few approaches that support (re)running the test suite automatically every time a file is saved in the IDE [29,18]. However, *test selection* as such is traditionally not performed: it is always the complete test suite that is run, including irrelevant tests, leading to an execution overhead that is larger than it actually needs to be.

For that reason, developers often manually select a few tests that seem appropriate, run them explicitly, and wait for feedback. The manual, regular, and explicit selection and execution of tests requires considerable discipline. Moreover, success is guaranteed only if no relevant test cases are omitted in the selection. A solution that automatically selects test cases to be executed in the background based on the applied changes to source code is preferable.

A. Sillitti et al. (Eds.): XP 2010, LNBIP 48, pp. 132–146, 2010.
© Springer-Verlag Berlin Heidelberg 2010

Approaches to test case selection are established: *Selective regression testing* [26] has long been a subject of research. Selective regression testing is concerned with reducing the set of tests that need to be executed to detect failures caused by recent modifications to the code base. However, researchers have not yet investigated the potential of integrating this technique into an IDE and having selected tests execute continuously in the background.

We suggest to select and execute tests *automatically* whenever the code status demands this. More precisely, it would be desirable to have support for TDD that, whenever source code is changed, *automatically executes exactly those tests that are affected by the actual modification*, giving developers *instant feedback* on whether the applied change breaks something or not.

In this paper, we propose our approach to *continuous selective testing* (CST) and present an implementation thereof in Squeak Smalltalk[1] [21]. Using an implementation of the suggested approach, developers will be supported as follows:

- Sets of relevant tests are selected based on dynamic analysis during the regular execution of tests.
- Relevant tests are executed continuously in the background after every modification to the code base.
- Developers are instantly informed about places in code that, resulting from an applied change, are no longer covered by tests.
- The introduction of new defects is made apparent immediately, which in turn lets developers focus on problems right away.

With that, our approach significantly improves on the way IDE tools provide immediate feedback in a development process adopting TDD. The main contributions of this paper are as follows:

- We present continuous selective testing as an approach relieving developers from the burden to select and run tests explicitly.
- We describe how test case selection in general can benefit from the internal program representation already available in IDEs and how differencing of two versions of a program can be avoided.
- We describe our approach to test case selection based on dynamic analysis, being not limited to statically-typed languages.

The remainder of this paper is organized as follows. The next section briefly summarizes TDD and presents the state of the art in tool support for it, providing further motivation for CST, which is presented and evaluated in Secs. 3 and 4. Related work is discussed in Sec. 5; Sec. 6 summarizes the paper and outlines future work.

2 Background and Motivation

In this section, we briefly introduce the terms and concepts of TDD. We then discuss current practices of developing tests and application code in accordance

[1] `www.squeak.org`

with TDD and point out the need for better tool support. Afterwards, we introduce the concepts of regression test selection and discuss current approaches.

2.1 The Three Phases of Test-Driven Development

Test-driven development distinguishes three phases of development [5]:

Red. Tests are written that specify new requirements on the system in an executable manner. When these new tests are run for the first time, failures or errors occur, as the system does not yet support the new requirements. An important guideline is to avoid writing application code if there is no test case that fails.

Green. The developer adds the required code to the system to make the failed test "green", i. e., run successfully. It is crucial that the developer write only code essential to the test in question. A successful test signals that the developer is done implementing the new requirement. It might happen that no code has to be added to make the test green, as the system already covers the newly defined requirement.

Refactor. The developer refactors towards the simplest design they can imagine. By definition of refactoring [13], new functionality must not be added during this phase. The tests can ensure that all required and specified features work after a refactoring. Running tests after each and every little change helps to avoid breaking features and provides instant feedback.

We can observe that tests and the regular execution of tests play an important role when developers employ the principles of TDD.

2.2 Tool Support for Test-Driven Development

Best practices in working with tests suggest to make only small changes and run tests immediately afterwards to get feedback. This suggestion is based, amongst others, on the following observations:

- Implementing new application functionality is a very complex activity. As every single step is inherently fault-prone, regular feedback is essential for detecting faults.
- Modifying source code without breaking existing functionality is also difficult. Adapting source code to new requirements or refactoring source code to a simpler design requires very detailed understanding, which to acquire is hard since source code abstracts from concrete execution paths. Having tests covering all parts of the respective code entities and running these tests regularly helps to detect faults early.
- The more steps are passed without getting feedback, the more difficult locating the source of a fault becomes. When a couple of source code entities are changed without running tests, and one ore more tests fail later on, isolating the modification that has caused the failure is not straightforward. Typically, developers are unaware of the complete set of modifications done before running the tests. Moreover, multiple failures might have different causes and

combinations of modifications might lead to completely unexpected behavior. To locate the defects, developers can revert modifications step by step or debug the current version. Both ways are tedious and time-consuming.

Running tests often and regularly helps developers to detect faults early, reduces the time required to localize defects, and gives confidence for the next adaptions and refactorings. However, running tests as often and regularly as suggested requires much discipline.

The necessary discipline is sometimes hard to bring up, for apprentices as well as experts. It is all too easy to ignore TDD theory, though well-understood and accepted, and continue modifying code without running tests. It is not necessarily only external factors, such as project schedules, that influence such decisions, but also internal ones like the strong will to finish a task. These aspects contradict with the required discipline.

Another issue with the theory of testing and test-first development is the implicitness of the relationship between test cases and application code they cover. When code is refactored or new features are implemented, existing code has to be modified. However, while developers are aware of recently implemented tests, they cannot know the set of all tests relying on a particular method. Hence, developers do not know the set of tests to be executed after a modification of a particular method. Consequently, all tests should be run after each modification, which is, however, increasingly time-consuming as projects grow. As a result of this, developers run only some tests regularly and the suite of tests is rarely executed, e. g., during integration builds.

Both aspects discussed above, the implicitness of the relationship between test cases and application code as well as the discipline required to run tests after each modification, question the usefulness of tests and test-first development. Our work provides tool support for TDD that alleviates these limitations and strengthens the benefits of testing.

3 Continuous Test Queuing, Selecting, and (Re-)Executing

In this section, we describe our approach to continuous selective testing called CST. It enables the continuous execution of selected tests directly after code modifications. Such automation relieves developers from the burden of executing tests manually. Selecting a subset of all tests and omitting those that cannot reveal faults reduces execution time and helps to provide feedback instantly. We have implemented the suggested approach in Squeak Smalltalk.

In the following, we will first introduce the concepts of regression test selection and then present the use of the IDE's program representation to detect and handle modifications to the code base. After that, we describe the queuing of tests and the selection and (re-)execution of tests according to the modification at hand. Finally, we present our extensions to the IDE providing instant feedback on test results.

3.1 Regression Test Selection

Regression testing refers to the practice of validating modified software; in particular, asserting that applied changes do not affect the software adversely [17]. The simplest approach to regression testing is to reuse the test suite used to exercise the previous version of the software. Fully running a large test suite can be unnecessarily costly, e. g., if only a few parts of the system were changed.

A technique to reduce the number of tests is *regression test selection*. It selects tests that have to be re-run to reveal a fault resulting from a particular change. Selecting an optimal set of tests is, however, generally inefficient [26]. Still, the set of tests traversing modifications can be computed efficiently. This set of *modification-traversing tests* can be considered a superset of the *fault-revealing tests* when the *Proper Regression Testing Assumption* [26] holds (P refers to a program and P' refers to the modified version of this program):

> When P' is tested with t, we hold all factors that might influence the output of P', except for the code in P', constant with respect to their states when we tested P with t.

A regression test selection technique is furthermore considered *safe* if it ensures to not omit tests revealing faults [17]. Several safe techniques have been proposed for purely procedural (e. g., [2,11,27]) as well as for object-oriented programming languages (e. g., [28,17]). Object-oriented programming is special as inheritance, polymorphism and thus late-binding have to be considered.

The most efficient and safe test selection technique is based on detecting modified code entities, such as functions or storage locations [26]. This technique was first implemented in TestTube [11] for software written in C. The technique is based on dynamic analysis [3]; test coverage information are recorded during each test run. For a new version of a software, the set of modified code entities can be detected. Based on coverage information, the technique selects and re-executes all tests that exercised the modified code entities in the previous version of the software. For object-oriented languages, the modified entity selection technique requires additional considerations due to language features such as inheritance and polymorphism enabling late binding.

Our approach, CST, is based on this technique of detecting modified code entities. CST records coverage information and selects tests on a method level. This procedure may select tests that do not traverse the modifications, because a test might only traverse unmodified parts of a method, for example. However, tracing on a more fine-grained level is much more expensive and does not pay off unless methods contain many control blocks [6].

3.2 Propagating Modifications to the Code Base

Most approaches to test selection are based on comparing the new with an earlier program version to detect change entities. Our approach takes advantage of an IDE's internal program representation. Fig. 1, on the left, depicts the setup of traditional approaches. IDE and test tools are not integrated and do not work

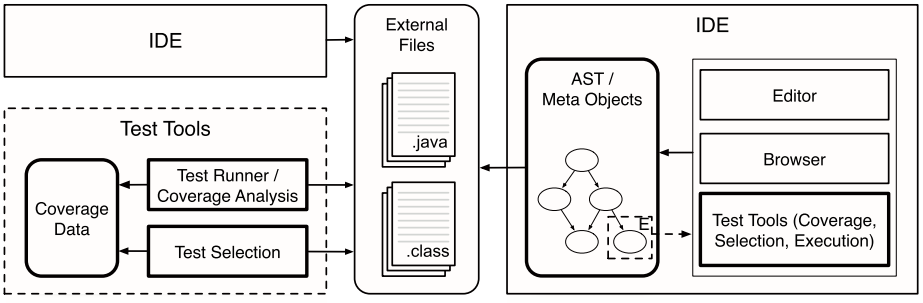

Fig. 1. The left-hand side shows a traditional setup where test selection tools and IDE work independently of each other. The right-hand side depicts CST integrating test selection into the IDE and taking advantage of the internal program representation.

together, each of them works rather separately on external program representations. In this setup, however, a test selection technique requires a comparison of program versions to detect modifications between two versions of a software. There exist differencing concepts and tool for both source code [1,17] and byte code [20].

We suggest to better integrate the tools for testing and test selection into the IDE as depicted on the right of Fig. 1. Every modification applied to the code base can produce an event notifying the IDE about the respective change. Using this notification mechanism, the test tools can process each modification to the code base. The tools are now able, for example, to automatically select and re-execute a set of test cases as necessary for the modification applied.

The set of events used to propagate code modifications to IDE tools has to be designed for the particular programming language and IDE, respecting the features of the language and the architecture of the IDE. In Squeak Smalltalk, for example, there are basically two operations to create or modify code objects. Sending a subclass-message to a class *c* creates a new or modifies an already existing subclass of class *c*. Sending the *compile:* message to a class object allows to compile a source code text of a method and puts it in the method dictionary of the corresponding class. Based on the effects of this two operations, the following change events can be defined for the Smalltalk [14] programming language, which is a rather simple language and does, for example, not provide any visibility modifiers; *class added*, *class removed*, *superclass changed*, *instance variable added*, *instance variable removed*, *method added*, *method modified*, and *method removed*. Note that class-specific ("static") state or behavior do not require special treatment as classes are also normal objects whose state and behavior are defined by meta-classes.

3.3 Queuing and Executing Tests for TDD

CST builds upon a well-defined set of different kinds of modification to the system. The event mechanism described above, with the possible modification

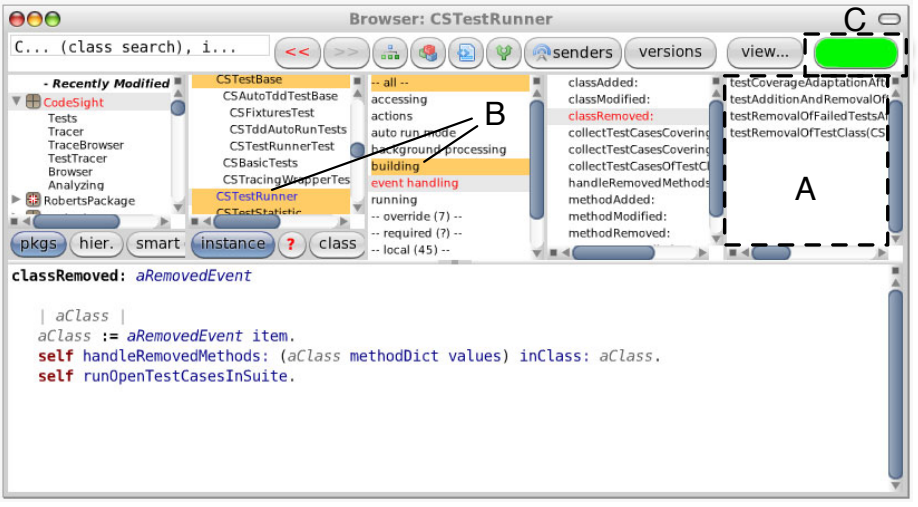

Fig. 2. An extended code browser in Squeak; having an additional panel on the right (A) that shows test cases covering the selected method named *classRemoved:*. Uncovered classes and methods are highlighted (B). A new widget (C) informs the developer on the current status of the test runner; whether it is currently running tests, and about the number of tests that have failed.

events it includes, allows for the continuous selection and execution of tests according to the current state of development.

Our approach distinguishes code entity modifications by their referal to test case code or non-test code. By convention, those methods of a class extending *TestCase* that are prefixed with *test* are treated as *test case methods*. Source code entities of test classes that are non-test methods, that is, attributes, *setUp*, *tearDown*, and other utility methods, are treated equally to application code.

When the creation of a test case method or modifications to one are reported, the developer is assumed to be in the red phase of the TDD cycle. The test runner will immediately execute the corresponding test case and provide instant feedback on the result. If the test fails, it will be queued. Failed test cases will be re-executed whenever a modification not related to a test method is reported. Now the developer is expected to be in the green or refactor phase, so the change has the potential to fix a test. All tests that still fail stay in the queue. A change of an entity can fix one or more tests cases, but the change can also introduce a fault that breaks other test cases. All test cases that might be affected by the reported change need to be re-executed. A technique to select the corresponding test cases is presented in the next subsection. The tests in the queue, failed before, are run first, providing earlier feedback on whether the current modification makes the failed test(s) pass.

To provide feedback on the test runs, we extended the tools for browsing and editing code. Whenever a modification is reported and the test runner executes

Table 1. Test selection procedures to be performed on entity modification events

Event	**Application Class**	***TestCase* Class**
class c removed	for each method of c, perform procedure for removing non-test methods (see below)	for each test and non-test method of c, perform procedure for removing respective methods (see below)
superclass of class c changed	re-run tests covering non-test methods in c and subclasses of c	see left; additionally, re-run tests defined in c and subclasses of c
Event	**Application Method**	**Test Method**
method m added to class c	re-run tests covering overridden methods with dynamic type c and tests covering overriding methods	run corresponding test case
method m removed	re-run covering tests	remove coverage links; remove from list of failed tests
method m modified	re-run covering tests	re-run corresponding test case

tests, a newly introduced GUI widget will inform the developer about the test runner's activities and the current status of the test result (Fig. 2). The widget turns red as soon as one test has failed. Tests are executed in a background process allowing the developer to navigate to the next code entity of interest and start editing it.

3.4 Re-executing Selected Tests for OO Software

The set of tests to be re-executed for an applied change should be minimized. CST relies on collecting test coverage information, and using this information to select tests that might be affected by a modification.

Using this coverage information of previous test runs, the CST tools can determine the set of tests that is to be re-executed for any reported change. The algorithms for the different kinds of changes are provided in table 1. Selecting the test cases that might be affected by a reported change is a two-step procedure:

1. If a non-test method is modified, the test runner collects and re-executes all test cases that covered this method previously. Therefore, the test runner can simply navigate the coverage relationship between the corresponding method objects.
2. CST also deals with modifications such as adding a method or changing the superclass that might affect late-bound method invocations. When, for example, an application method m' is added to a class c', and m' overrides a method m in a superclass c, CST will execute tests that have covered m'. More precisely, it will select those tests that previously exercised m for instances of c.

As mentioned above, the set of meaningful events, which reports modified code entities, may vary between languages providing different sets of features. The

algorithms to be applied to determine a safe set of tests may vary as well. If the language supports multiple inheritance, for example, the algorithms have to consider the possibility of multiple superclasses and the respective linearization order applied to method dispatch.

As pointed out in [17], a safe test selection technique for object-oriented software must also consider exception handling. CST allows to consider exceptions similarly to other code entities. A basic method constructing an exception object needs to be instrumented; for instance, default constructors in Java, or *basicNew* in Smalltalk. Using the receiver's dynamic type recorded for each method call, we can determine whether an exception was created and thrown during the execution of a test case. If the exception class hierarchy is changed, all test cases that might be affected can be identified easily.

3.5 Establishing a Coverage Relationship

Test coverage information used for test selection is collected during regular test execution. We decided to collect this information only for packages and classes of interest. This typically excludes basic development classes such as the collection or system libraries. The selection of relevant packages and exclusion of others avoids unnecessary overhead [15]. To record method coverage information, we use method wrappers [8]. Actual method code is wrapped in tracing code that records the call of the wrapped method in the context of the currently running test case, and forwards the sent message to the wrapped method afterwards.

Test coverage information is integrated into the IDE's program representation. In CST, we establish and maintain a coverage relationship between test case methods and methods covered during test execution, as depicted in Fig. 3. Here, we generally refer to objects representing methods in the IDE; Squeak Smalltalk provides so-called *CompiledMethod* objects to reflect upon and work with methods in the system.

Employing the test-first principle and using CST, tests run frequently and the coverage relationship has to be maintained for test runs. To avoid unnecessary

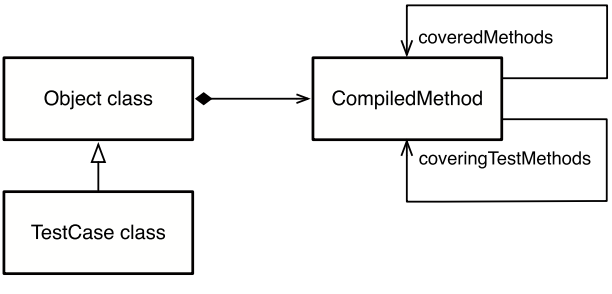

Fig. 3. The coverage relationship between test methods, included in *TestCase* classes, and application methods covered by them

start-up costs, tracing logic is installed incrementally after each compilation step. When the developer selects packages and classes of interest, wrapper logic is initially installed. If source code entities matching the selection criteria are added, they are wrapped directly after creation. This incremental approach avoids the need to instrument source code for each test run.

Using CST, developers can also be provided with instant feedback regarding test coverage. Classes and methods that are not covered any more are highlighted in the code browser (Fig. 2). The feedback supports developers in ensuring high method coverage. We further extended the code browser with an additional fifth panel (Fig. 2) that shows all test cases covering the currently selected method. This extension makes the coverage relationship visible and the applied test selection technique transparent for developers.

4 Evaluation

In the first part of this section, we describe our experience using CST, and its implementation in Squeak Smalltalk. After that, we report on our experiments for gaining insights into test set reduction.

4.1 Using CST in Developing CST

From early on, we used our CST tools to develop their next versions. This bootstrapping allowed us to get feedback on both the suitability of our approach and the quality of our implementation.

Unsurprisingly and as expected, we made mistakes during development and introduced defects into our code base in both unit tests and units under test, leaving us with both false positives and false negatives. Here, our tool served its purpose well by making us aware of unexpected results immediately after each method save. With that, and even if the problem itself was sometimes hard to understand, the cause of the problem becoming apparent was easily recognized as *the last change done to the system.*

Using our tools revealed another benefit in the form of obtaining instant feedback on test *coverage* after modifications. Getting this information right away helped to better understand the dynamics of our system and to remove code that was not needed any longer.

4.2 Test Set Reduction

We were interested in how effectively CST reduces the amount of tests to be run when a method is modified. We therefore conducted experiments on the following systems:

CST. The implementation of CST for Squeak Smalltalk (cf. Sec. 4.1).
XP-Forums. A groupware[2] supporting collaboration on artifacts specific to distributed agile software development.

[2] `http://www.hpi.uni-potsdam.de/swa/projects/xpf`

AweSOM. A virtual machine[3] for a Smalltalk dialect, implemented in Squeak.
Seaside 2.8. A Smalltalk-based Web framework.[4]

For these four very different software systems, we determined the number of tests covering each method. These tests have to be executed when the corresponding method is modified. The results and the overall number of tests for the systems are presented in Table 2. The comparison shows that method coverage analysis can significantly reduce the number of tests to be executed. It is also interesting to see that some methods of the systems are only covered by one test whereas other methods are covered by all tests.

Our test selection technique is the first to record dynamic type information (DTI) to reduce the set of tests in case of subclass modifications. To evaluate our assumption that collecting this information is useful, we performed the following experiment on the systems:

1. Collect those application methods whose classes have at least one subclass, and which are not overridden.
2. Determine how many tests must be re-run in case a method is overridden in one of its class's subclasses.

The results, also presented in Table 2, show that using run-time information about the actual receivers of a message can significantly reduce the test set size. We conclude that our test selection technique is both safe, by considering late binding in OOP, and still very effective.

Table 2. Results of evaluting the effectiveness of test set reduction on four projects

Project	Tests	**Cov. Tests per Method**			**Savings Using DTI in %**		
		min	max	median	min	max	median
CST	55	1	55	15	32	100	86
XP-Forums	98	1	95	24	0	100	38
AweSOM	124	1	124	23	3	100	97
Seaside	183	1	75	3	3	100	50

5 Related Work

In this section, we discuss related work on two major aspects considered in this paper; tool support for TDD in general and regression test selection.

5.1 Tool Support for Test-Driven Development

The Ruby and Ruby on Rails community in particular is very committed to agile methodologies. The community has reported the combination of several tools[5],

[3] `http://www.hpi.uni-potsdam.de/swa/projects/som`
[4] `http://www.seaside.st`
[5] `http://www.zenspider.com/ZSS/Products/ZenTest/`

such as Growl and autotest/autospec, to run tests when a file is saved and notify developers on test results. Coverage data are collected to reduce the set of tests to be executed. The applied test selection technique considers modifications on the granularity of files, thus it may select more than a modified-entity technique.

The authors of [24] present an approach to guide TDD which is complementary with our approach. TDD-Guide is an IDE-integrated tool that guides developers in applying TDD. Based on a set of rules, which can be adapted, the tool processes IDE information about the current development status and informs developers about compliance with rules; for example, whether they comply with adding new functionality only if a failing test exists.

The *continuous testing* approach [29] is especially interesting as it is close to our intentions. However, we observe notable differences. First of all, it does not select regression tests but only prioritizes them to allow for a more efficient test execution and more timely feedback. Change analysis focuses on files instead of single methods. Conversely, we do restrict the set of test cases to be executed to those actually covering changed methods.

5.2 Regression Test Selection

Regarding regression test selection, we restrict the discussion of related work to approaches that support object-oriented programming languages and constructs. Among these, we identify three dimensions of interest. First, approaches can be based on source code or binary formats. Second, static or dynamic analysis can be applied. Third, the granularity of application entities (files, classes, methods, single statements or expressions) is relevant. The selection strategy in CST is based on the *binary* format of compiled-method representations, applies *dynamic* analysis, and is *fine-grained* in that single methods are the units of analysis. To the best of our knowledge, CST is the first approach to combine these features.

A large family of approaches applies call graph analysis using source code [4,25,28,17,12,1] or a binary format [10,30,20] of the software. Dedicated mechanisms for object-oriented features, e. g., related to selecting test cases for subclasses, have also been devised [9,16,23,19,7].

The above maintain an internal representation of the program that enables a detailed and correct comparison. Call graphs allow for a very fine-grained analysis, down to single statements. Differencing algorithms in these approaches rely on static source code analysis, effectively restricting the approach to statically typed languages. Due to applying dynamic code coverage analysis, CST supports dynamically-typed languages.

Call graph-based approaches are also more precise than CST, which works at the granularity of single methods, but they are also more expensive [6] (cf. Sec. 3.1). Generally, good object-oriented programming style suggests to use small methods and few control structures—the liabilities our less fine-grained analysis brings about are likely to go unnoticed if these practices are applied.

One of the binary-format approaches mentioned above [20] is also fine-grained, regarding single methods as analysis units. The main difference to CST is that the latter applies dynamic analysis to extract actual method coverage data.

Regarding the *safety criterion* [26,17], we can report that CST is safe, since coverage analysis as applied therein selects all tests covering changes. In addition, object-oriented constructs and late binding are honored. Finally, we regard message receiver type information to limit the set of tests to execute upon subclass changes.

6 Summary and Outlook

We have presented CST, our approach to *continuous selective testing* and an implementation thereof in Squeak Smalltalk. Using a tool such as CST relieves developers from selecting and executing tests manually. Based on the actual modification, a selected set of tests is executed transparently in the background, reporting instantly on the effect of the applied change with respect to the overall set of tests to be passed. CST also takes advantage of an IDE's program representations and thus avoids differencing to detect modified code entities. The test selection technique that we apply is based on dynamic analysis and thus does not require a statically typed language. It is the first approach to test selection that makes use of run-time type information to reduce the test set in case of subclass modifications.

Future work on CST most importantly includes the investigation of techniques to prioritize selected tests and the integration of an appropriate candidate. We expect opportunities to reveal faults and inform developers about them even faster.

Acknowledgments. We gratefully acknowledge the financial support of the Hasso Plattner Design Thinking Research Program for the project "Agile Software Development in Virtual Collaboration Environments".

References

1. Apiwattanapong, T., Orso, A., Harrold, M.J.: JDiff: A differencing technique and tool for object-oriented programs. Automated Software Engineering 14(1), 3–36 (2007)
2. Ball, T.: On the limit of control flow analysis for regression test selection. ACM SIGSOFT Software Engineering Notes 23(2), 134–142 (1998)
3. Ball, T.: The Concept of Dynamic Analysis. In: ESEC/FSE-7: Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering, London, UK, pp. 216–234. Springer, Heidelberg (1999)
4. Bates, S., Horwitz, S.: Incremental program testing using program dependence graphs. In: POPL 1993: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 384–396. ACM, New York (1993)
5. Beck, K.: Test-driven Development: By Example. Addison-Wesley Professional, Reading (2003)
6. Bible, J., Rothermel, G., Rosenblum, D.S.: A comparative study of coarse-and fine-grained safe regression test-selection techniques. ACM Transactions on Software Engineering and Methodology 10(2), 149–183 (2001)

7. Binder, R.: Testing object-oriented systems: models, patterns, and tools. Addison-Wesley, Reading (1999)
8. Brant, J., Foote, B., Johnson, R.E., Roberts, D.: Wrappers to the Rescue. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 396–417. Springer, Heidelberg (1998)
9. Cheatham, T.J., Mellinger, L.: Testing object-oriented software systems. In: CSC 1990: Proceedings of the 1990 ACM annual conference on Cooperation, pp. 161–165. ACM, New York (1990)
10. Chen, Y., Probert, R.L., Sims, D.P.: Specification-based regression test selection with risk analysis. In: CASCON 2002: Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research, p. 1. IBM Press (2002)
11. Chen, Y.F., Rosenblum, D. S., Vo, K.P.: TestTube: A system for selective regression testing. In: Proceedings of 16th International Conference on Software Engineering, 1994, ICSE-16, pp. 211–220 (1994)
12. Clarke, P., Malloy, B., Gibson, P.: Using a taxonomy tool to identify changes in OO software. In: Proceedings of Seventh European Conference on Software Maintenance and Reengineering, 2003, pp. 213–222 (2003)
13. Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley Professional, Reading (1999)
14. Goldberg, A., Robson, D.: Smalltalk-80: The Language and its Implementation. Addison-Wesley, Reading (1983)
15. Gschwind, T., Oberleitner, J.: Improving Dynamic Data Analysis with Aspect-Oriented Programming. In: CSMR 2003: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering, Washington, DC, USA, pp. 259–268. IEEE Computer Society, Los Alamitos (2003)
16. Harrold, M.J., McGregor, J.D., Fitzpatrick, K.J.: Incremental testing of object-oriented class structures. In: ICSE 1992: Proceedings of the 14th international conference on Software engineering, pp. 68–80. ACM, New York (1992)
17. Harrold, M.J., Jones, J.A., Li, T., Liang, D., Orso, A., Pennings, M., Sinha, S., Spoon, S.A., Gujarathi, A.: Regression Test Selection for Java software. In: Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 312–326. ACM, New York (2001)
18. Harrold, M.J., Orso, A.: Retesting Software During Development and Maintenance. In: Frontiers of Software Maintenance, FoSM 2008, pp. 99–108 (2008)
19. Hsia, P., Li, X., Chenho Kung, D., Hsu, C.T., Li, L., Toyoshima, Y., Chen, C.: A technique for the selective revalidation of OO software. Journal of Software Maintenance: Research and Practice 9(4) (1997)
20. Huang, S., Chen, Y., Zhu, J., Li, Z.J., Tan, H.F.: An optimized change-driven regression testing selection strategy for binary Java applications. In: Proceedings of the 2009 ACM symposium on Applied Computing, pp. 558–565. ACM, New York (2009)
21. Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., Kay, A.: Back to the Future: the Story of Squeak, a Practical Smalltalk Written in Itself. In: Proc. OOPSLA 1997, pp. 318–326. ACM Press, New York (1997)
22. Beck, K., Andres, C.: Extreme Programming Explained: Embrace Change, 2nd edn. Addison-Wesley Longman, Amsterdam (2004)
23. Kung, D.C., Gao, J., Hsia, P., Toyoshima, Y., Chen, C.: On regression testing of object-oriented programs. The Journal of Systems & Software 32(1), 21–40 (1996)
24. Mishali, O., Dubinsky, Y., Katz, S.: The TDD-guide training and guidance tool for test-driven development. In: The International Conference on Agile Processes and

eXtreme Programming in Software Engineering (XP), Limerick, Ireland, Springer, Heidelberg (2008)
25. Rothermel, G., Harrold, M.J.: A safe, efficient algorithm for regression test selection. In: ICSM '93: Proceedings of the Conference on Software Maintenance, Washington, DC, USA, pp. 358–367. IEEE Computer Society Press, Los Alamitos (1993)
26. Rothermel, G., Harrold, M.J.: Analyzing regression test selection techniques. IEEE Transactions on Software Engineering 22(8), 529–551 (1996)
27. Rothermel, G., Harrold, M.J.: A safe, efficient regression test selection technique. ACM Transactions on Software Engineering and Methodology (TOSEM) 6(2), 173–210 (1997)
28. Rothermel, G., Harrold, M.J., Dedhia, J.: Regression Test Selection for C++ Software. Software Testing, Verification & Reliability 10(2), 77–109 (2000)
29. Saff, D., Ernst, M.D.: Reducing wasted development time via continuous testing. In: ISSRE 2003: Proceedings of the 14th International Symposium on Software Reliability Engineering, Washington, DC, USA, p. 281. IEEE Computer Society Press, Los Alamitos (2003)
30. Zheng, J., Robinson, B., Williams, L., Smiley, K.: A process for identifying changes when source code is not available. In: MPEC 2005: Proceedings of the second international workshop on Models and processes for the evaluation of off-the-shelf components, pp. 1–4. ACM, New York (2005)