CSlicerCloud: A Web-Based Semantic History Slicing Framework

Yi Li
University of Toronto
liyi@cs.toronto.edu

Chenguang Zhu
University of Texas at Austin
cgzhu@utexas.edu

Julia Rubin
University of British Columbia
mjulia@ece.ubc.ca

Marsha Chechik
University of Toronto
chechik@cs.toronto.edu

ABSTRACT

Traditional commit-based sequential organization of software version histories is insufficient for many development tasks which require high-level, semantic understanding of program functionality, such as porting features or cutting new releases. *Semantic history slicing* is a technique which uses well-organized unit tests as identifiers for corresponding software functionalities and extracts a set of commits that correspond to a specific high-level functionality. In this paper, we present CSlicerCloud, a Web-based semantic history slicing service tailored for Java projects hosted on GitHub. It is accessible through Web browsers and powered in the backend by a collection of history slicing techniques underneath. We evaluated CSlicerCloud on a dataset containing developer-annotated change histories collected from 10 open source software projects. A video demonstration which showcases the main features of CSlicerCloud can be found at https://youtu.be/7kcswA0bQzo.

KEYWORDS

Version histories, software evolution, program semantics

ACM Reference Format:
Yi Li, Chenguang Zhu, Julia Rubin, and Marsha Chechik. 2018. CSlicerCloud: A Web-Based Semantic History Slicing Framework. In *ICSE '18 Companion: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3183440.3183491

1 INTRODUCTION

Software Configuration Management systems (SCMs) are widely used in software development practices. These systems, such as Git [4] and SVN [21], are useful for recording incremental changes made by developers, examining or reverting changes, identifying developers responsible for a specific change, and more. Incremental changes are manually grouped by developers to form *commits*. Commits are stored sequentially and ordered by their time stamps, so that it is convenient to trace back to any version in the history. To facilitate parallel development, branches are used, for example, to store a still-in-development prototype version of a project or multiple project variants targeting different customers.

However, the text-based organization of changes lacks support for many tasks that require high-level, semantic understanding of program functionality [13, 16]. For example, developers often need to locate and transfer functionality from one branch to another, either for porting bug fixes or for splitting large chunk commits into multiple functionally-independent pull requests. Several SCM systems provide mechanisms of "replaying" commits on a different branch, e.g., the `cherry-pick` command in Git. Yet, little support is provided for matching high-level functionality with commits that implement it: SCM systems only keep track of temporal and text-level dependencies between the managed commits. The job of identifying the exact set of commits implementing the functionality of interest is left to the developers.

Motivated by these challenges, several *semantic history slicing* techniques [10–12] have been recently proposed. A *semantic history slice* [12] is a set of related changes in the original history which preserve the target test behaviors (a.k.a. the slicing criteria). In this paper, we describe a Web-based semantic history slicing framework, CSlicerCloud, which provides users with a flexible and intuitive way to interact with the underlying history slicing techniques and implements a number of optimizations to improve both the quality and the efficiency of slicing. The tool and more detailed information are available at: http://www.cs.toronto.edu/~liyi/cslicer.

We see applications of CSlicerCloud in many different software evolution management scenarios.

A1. Porting Functionalities Across Versions. Often, a developer works on multiple functionalities at the same time which could result in mixed commit histories concerning different issues. However, when submitting pull requests for review, contributors should refrain from including unrelated changes as suggested by many project contribution guidelines. Despite the efforts of keeping the development of each issue on separate branches, isolating each functional unit as a self-contained pull request is still a challenging task. For a particular pull request, the test cases created for validation can be used as slicing criteria to identify relevant commits from the developers' local histories in their forked repositories. CSlicerCloud can identify the set of commits required for back-porting a functionality to earlier versions of a software project.

A2. Creating Pull Requests. Even in very disciplined projects, when such commits can be identified by browsing their associated log messages, the functionality of interest might depend on earlier commits in the same branch. To ensure correct execution of the desired functionality, all dependencies have to be identified and migrated as well, which is a tedious and error-prone manual task. Given test cases for the functionalities to be ported, CSlicerCloud can automatically compute the required changes and, at the same time, effectively avoid including unnecessary changes, i.e., it can facilitate creation of logically clean and easy-to-merge pull requests.

A3. Cutting New Releases. Many software projects periodically cut releases by cherry-picking commits corresponding to desired functionalities from the develop branch to the release branches, e.g., Bazel [1]. CSlicerCloud analyses the change history since the last release and uses feature test cases to locate the desired set of commits which guarantee not to cause merge conflicts and pass the tests when moved to the release branch.

Organization. The rest of this paper is organized as follows: In Sec. 2, we overview the architecture and user interface of CSlicerCloud, illustrating it by an example use case A3. In Sec. 3, we describe the back-end implementations of CSlicerCloud which unifies a number of history slicing algorithms and enables several optimizations, such as caching of slicing results. In Sec. 4, we evaluate the performance and effectiveness of CSlicerCloud on a dataset [24] of change histories collected from 10 open source software projects. In Sec. 5, we discuss related work. We conclude in Sec. 6.

2 OVERVIEW OF CSLICERCLOUD

In this section, we overview the architecture of CSlicerCloud and demonstrate its user interface on an example use case (A1).

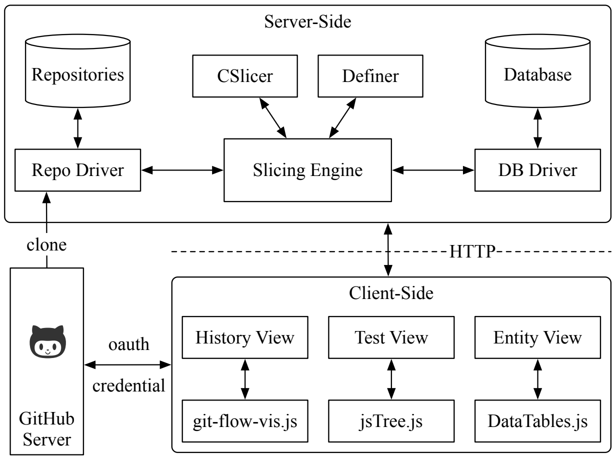
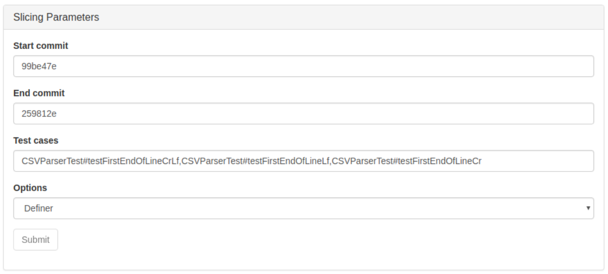


Figure 1: Architecture of CSlicerCloud.

Architecture. CSlicerCloud consists of a web-based front-end, implemented on top of the Node.js JavaScript runtime [15], and a server-side back-end. See Fig. 1 for an architectural overview. The front-end obtains user credentials from the GitHub server through the OAuth protocol [6] and uses them to authenticate users and access their repository meta data. The front-end also collects user requests and communicates to the back-ends via HTTP connections. The back-end retrieves repository data from the GitHub server; the slicing engine performs history slicing as per the users' requests, indexes version histories and caches slicing results, storing them in the database; finally, the results are communicated back to the front-end. Currently, the slicing engine supports two different slicing algorithms: CSlicer [10] and Definer [11].

The user interface of CSlicerCloud consists of four components – the *slicing parameter panel*, *history view*, *test view*, and *entity view*.

Imagine that a developer wants to migrate a feature – "Adding a placeholder in the Lexer and CSV parser to store the end-of-line string" which was introduced in version 1.5 of the Apache Common CSV project [3] – to another branch, but he/she is unsure which commits are required for the target feature to work correctly on the target branch.


Slicing Parameters

Start commit

99be47e

End commit

259812e

Test cases

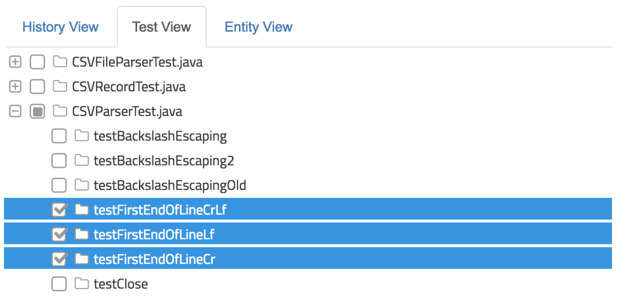
CSVParserTest#testFirstEndOfLineCrLf,CSVParserTest#testFirstEndOfLineLf,CSVParserTest#testFirstEndOfLineCr

Options

Definer

Figure 2: Setting slicing parameters: start commit, end commit, test cases, and slicing algorithm.

Slicing Parameter Panel. In order to use CSlicerCloud for identifying the commits required, a user must specify a number of slicing parameters. Fig. 2 shows the slicing parameter panel of CSlicerCloud. A user defines a history range by specifying the "start commit" and the "end commit", in this case, the commits "`#99be47e`" and "`#259812e`", respectively. This can also be done by directly clicking on the commits in the history view.



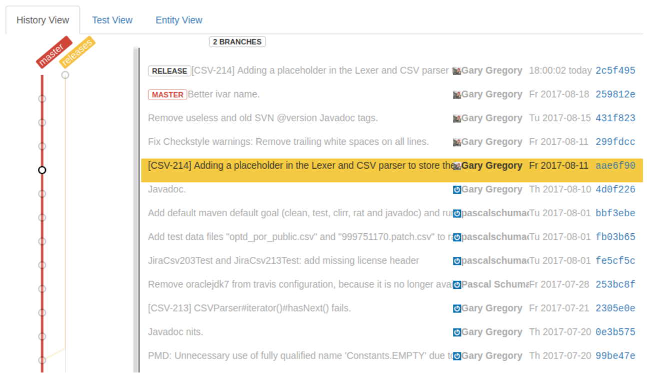
- CSVFileParserTest.java
- CSVRecordTest.java
- CSVParserTest.java
  - testBackslashEscaping
  - testBackslashEscaping2
  - testBackslashEscapingOld
  - ✓ testFirstEndOfLineCrLf
  - ✓ testFirstEndOfLineLf
  - ✓ testFirstEndOfLineCr
  - testClose

Figure 3: The test view of CSlicerCloud.

Test View. Then, a set of feature test cases needs to be provided as the slicing criteria. The test view visualizes the unit tests of a Java project, and a user can select the tests he/she wants to use for history slicing tasks. Fig. 3 gives a screenshot of the test view showing the test cases organized in an expandable tree structure. Each top-level node represents a Java test file, e.g., "`CSVParserTest.java`". The leaf nodes become visible when a test file node is expanded. They represent the test methods contained in the corresponding test file. For instance, there are three test cases associated with the target feature, "`testFirstEndOfLineCrLf`", "`testFirstEndOfLineLf`", and "`testFirstEndOfLineCr`". Each method also comes with a check box which allows it to be included as one of the slicing criteria.

History View. The history view visualizes the version histories of a given software repository and allows users to define a history range for the subsequent history slicing tasks. Fig. 4 shows a screenshot of the history view displaying the version histories of the Common CSV project. The left pane depicts the branches and commits graphically while the right pane lists other metadata including log messages, authors, and SHA-1 commit IDs of the corresponding commits. Each item in the commit list is clickable such that a history range can be defined by selecting a starting and an ending commit.

When the history slicing process is finished, the slicing results are shown in the history view; in our case, the commit `#aae6f90` is highlighted. CSlicerCloud can also create a new branch with the computed history slice as a migration dry-run (see Fig. 4).



RELEASE [CSV-214] Adding a placeholder in the Lexer and CSV parser	Gary Gregory	18:00:02 today	2c5f495
MASTER Better ivar name.	Gary Gregory	Fr 2017-08-18	259812e
Remove useless and old SVN @version Javadoc tags.	Gary Gregory	Tu 2017-08-15	431f823
Fix Checkstyle warnings: Remove trailing white spaces on all lines.	Gary Gregory	Fr 2017-08-11	299fdcc
[CSV-214] Adding a placeholder in the Lexer and CSV parser to store the	Gary Gregory	Fr 2017-08-11	aae6f90
Javadoc.	Gary Gregory	Th 2017-08-10	4d0f226
Add default maven default goal (clean, test, clirr, rat and javadoc) and rui	pascalschuma	Tu 2017-08-01	bbf3ebe
Add test data files "optd_por_public.csv" and "999751170.patch.csv" to r	pascalschuma	Tu 2017-08-01	fb03b65
JiraCsv203Test and JiraCsv213Test: add missing license header	pascalschuma	Tu 2017-08-01	fe5cf5c
Remove oraclejdk7 from travis configuration, because it is no longer avai	Pascal Schuma	Fr 2017-07-28	253bc8f
[CSV-213] CSVParser#iterator()#hasNext() fails.	Gary Gregory	Fr 2017-07-21	2305e0e
Javadoc nits.	Gary Gregory	Th 2017-07-20	0e3b575
PMD: Unnecessary use of fully qualified name 'Constants.EMPTY' due to	Gary Gregory	Th 2017-07-20	99be47e

Figure 4: The history view of CSlicerCloud. Slicing results are highlighted.

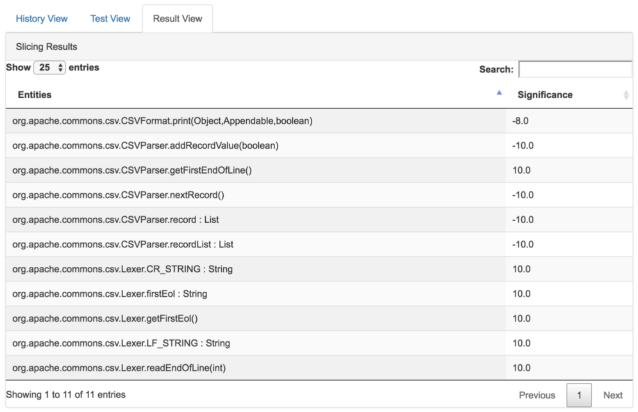
The graphical visualization of Git version histories is powered by the JavaScript library GitFlowVisualize [5]. GitFlowVisualize accepts commit and branch data as JSON [9] objects parsed from the target project Git repositories. We have modified the library to enable history selection and result highlighting.

Entity View. The entity view displays the code entity-level *significance scores* [11] inferred from running Definer for all code entities changed within the analysis range. The significance score indicates the estimated relative importance of a code entity with respect to the target tests for a particular input history. Fig. 5 shows an example entity view which displays the significance scores in a table. For instance, the third row in the table shows that the method "`CSVParser.getFirstEndOfLine()`" has a significance score of 10.0. This indicates the relative importance; absolute value carries little meaning other than a positive score indicates that the changes on the entity can potentially affect the test results. The entities with negative scores are less likely to impact the tests.

3 BACK-END IMPLEMENTATION

The server-side back-end consists of three components: the *slicing engine*, the *DB driver* and the *Repo driver*.

Slicing Engine. The slicing engine is the core of the server-side. It acts as a wrapper of the underlying history slicing techniques, dispatching slicing job requests coming from multiple users with various configuration parameters. When idle, the slicing engine can index cached version histories in order to speed up future history



Entities	Significance
org.apache.commons.csv.CSVFormat.print(Object,Appendable,boolean)	-8.0
org.apache.commons.csv.CSVParser.addRecordValue(boolean)	-10.0
org.apache.commons.csv.CSVParser.getFirstEndOfLine()	10.0
org.apache.commons.csv.CSVParser.nextRecord()	-10.0
org.apache.commons.csv.CSVParser.record : List	-10.0
org.apache.commons.csv.CSVParser.recordList : List	-10.0
org.apache.commons.csv.Lexer.CR_STRING : String	10.0
org.apache.commons.csv.Lexer.firstEol : String	10.0
org.apache.commons.csv.Lexer.getFirstEol()	10.0
org.apache.commons.csv.Lexer.LF_STRING : String	10.0
org.apache.commons.csv.Lexer.readEndOfLine(int)	10.0

Showing 1 to 11 of 11 entries

Figure 5: The entity view of CSlicerCloud.

slicing computations. The significance score of changes with respect to a test case can also be precomputed and stored for future use.

Repo Driver. The Repo driver hides the low-level details of repository operations and provides an abstracted interface for the slicing engine. The supported high-level operations include cloning repositories from the GitHub server, compiling source code with Maven, retrieving test cases, running tests with Surefire [20] and processing test results. The Repo driver runs asynchronously, with callbacks for time-consuming operations. This prevents the progress of the clients from being blocked.

In order to extract the list of available test cases and display it in the test view, we first check out the repository to the selected "end commit". We then compile the tests into byte code and use the BCEL [2] library to analyze the byte code and output the fully qualified test method names as JSON objects.

DB Driver. The DB driver supports writing to and querying from the database which maintains the "*runs*" and "*significance*" tables. They record the parameters and the results of each history slicing run, and the significance scores of all analyzed code entities, respectively. If a subsequent slicing request matches one of the entries in the database, the cached result is immediately returned. Otherwise, the significance scores of the relevant entities are queried from the "significance" table to initialize a new slicing run.

4 EVALUATION

In this evaluation, we used 98 functionality-history pairs from 10 open source projects from the DoSC dataset [24] where each of the functionalities came with a set of tests, a set of commits annotated by developers in log messages as related to functionalities. We then ran CSlicerCloud with Definer chosen as the underlying slicing algorithm to compute a semantic history slice and compared it with the commits labeled by developers. The studies were conducted on a desktop computer running Linux with an Intel i7 3.4GHz processor and 16GB of RAM. All results of CSlicerCloud were verified to be *1-minimal* [11], meaning that removing a single commit would result in a set that no longer passes the tests.

Fig. 6 shows the results of our comparison. The vertical and horizontal axises represent the precision and recall of CSlicerCloud's results w.r.t. the developers' annotations. On most examples (75/98),

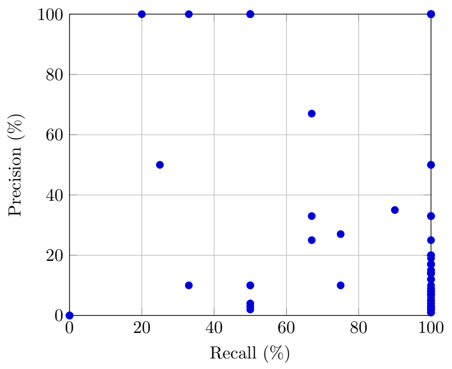


Figure 6: Precision and recall of CSlicerCloud w.r.t. developer annotations.

CSlicerCloud was able to find all the commits labeled by developers. Also, in the majority of these cases, CSlicerCloud found more commits which were missed by developers. Since the results produced by CSlicerCloud were already minimal, it shows that the developer-annotated commits are often inadequate for preserving the test behaviors.

CSlicerCloud can find fewer commits in some other cases, mainly due to the inadequacy of tests. There have been three examples where CSlicerCloud got zero precision and recall. After inspecting the documentation and the content of the commits, we found that in these cases, the target functionalities were aimed to improve performance instead of implementing new features. For example, CALCITE-1337[1] improves performance of an existing functionality and thus the labeled commits also affect performance but not the passing/failing of the associated tests.

5 RELATED WORK

Analysis and understanding software histories is an active area of research, aimed at retrieving useful information from change histories to help understand development practices [14, 19], localize bugs [17, 23], and support predictions [8, 25].

History slicing [19] and *history transformation* [7, 13] further aim to create flexible views of the change histories at varying granularities, instead of the fixed commit-based representation, to better facilitate the specific software evolution task at hand. For example, Muşlu et al. [13] introduced several history transformation operators such as Collapse, Expand and Move which manipulate changes and produce customized history views. Our proposed approach can be used in combination with these operations to create high-level semantic views of the histories.

Delta debugging [23] uses divide-and-conquer-style iterative test executions to narrow down potential causes of software failures. This problem can be considered as a special case of semantic history slicing: given a series of changes which cause a program failure, the goal is to locate the minimal cause of the failure, or, in other words, the changes which preserve the failure-inducing property. Traditional bug localization techniques [18, 22], based on information retrieval, take a bug report as input and identify source code files/methods that need to be fixed. It is possible to extend current bug localization solutions to use semantic history slicing for identifying problematic commits. This would provide additional traceability to the bug localization results and enable more accurate issue management.

[1]https://issues.apache.org/jira/browse/CALCITE-1337

6 CONCLUSION

In this paper, we described the architecture of CSlicerCloud, its user interface, prominent optimizations, and showed the effectiveness of our tool by evaluating it on a dataset collected from open source software projects. The flexible architecture of CSlicerCloud enables the integration of many different underlying history slicing algorithms, and its Web-based interface allows users to easily access the history slicing services within a browser.

ACKNOWLEDGEMENTS

We thank Shayan Kousha for his help with implementing the front-end. The work was supported in part by NSERC.

REFERENCES

[1] Bazel 2017. https://bazel.build/support.html#releases. (2017).
[2] BCEL 2015. Apache Commons Byte Code Engineering Library. https://commons.apache.org/proper/commons-bcel. (2015).
[3] CSV 2017. Using Apache Commons CSV. https://commons.apache.org/proper/commons-csv. (2017).
[4] Git 2016. Git Version Control System. https://git-scm.com. (2016).
[5] GitFlow 2017. GitFlowVisualize. https://www.npmjs.com/package/git-flow-vis. (2017).
[6] Dick Hardt. 2012. *The OAuth 2.0 Authorization Framework*. RFC 6749. RFC Editor, Fremont, CA, USA. http://www.rfc-editor.org/rfc/rfc6749.txt
[7] Shinpei Hayashi, Takayuki Omori, Teruyoshi Zenmyo, Katsuhisa Maruyama, and Motoshi Saeki. 2012. Refactoring Edit History of Source Code. In *Proc. of ICSM'12*. 617–620.
[8] Kim Herzig and Andreas Zeller. 2013. The Impact of Tangled Code Changes. In *Proc. of MSR'13*. Piscataway, NJ, USA, 121–130.
[9] JSON 2017. Introducing JSON. http://www.json.org. (2017).
[10] Yi Li, Julia Rubin, and Marsha Chechik. 2015. Semantic Slicing of Software Version Histories. In *Proc. of ASE'15*. 686–696.
[11] Yi Li, Chenguang Zhu, Julia Rubin, and Marsha Chechik. 2016. Precise Semantic History Slicing through Dynamic Delta Refinement. In *Proc. of ASE'16*. 495–506.
[12] Yi Li, Chenguang Zhu, Julia Rubin, and Marsha Chechik. 2017. Semantic Slicing of Software Version Histories. *IEEE Transactions on Software Engineering* (2017).
[13] Kivanç Muşlu, Luke Swart, Yuriy Brun, and Michael D. Ernst. 2015. Development History Granularity Transformations. In *Proc. of ASE'15*. 697–702.
[14] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2012. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering* 38, 1 (Jan 2012), 5–18.
[15] NodeJS 2017. Node.js. https://nodejs.org. (September 2017).
[16] Julia Rubin, Andrei Kirshin, Goetz Botterweck, and Marsha Chechik. 2012. Managing Forked Product Variants. In *Proc. of SPLC'12*. 156–160.
[17] Ripon Saha and Milos Gligoric. 2017. Selective Bisection Debugging. In *Proc. of FASE'17*. Springer-Verlag New York, Inc., 60–77.
[18] Ripon K. Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E. Perry. 2013. Improving Bug Localization Using Structured Information Retrieval. In *Proc. of ASE'13*. 345–355.
[19] Francisco Servant and James A. Jones. 2011. History Slicing. In *Proc. of ASE'11*. 452–455.
[20] Surefire 2017. Maven Surefire Plugin. http://maven.apache.org/surefire/maven-surefire-plugin. (2017).
[21] SVN 2016. Apache Subversion (SVN) Version Control System. http://subversion.apache.org. (2016).
[22] Shaowei Wang and David Lo. 2016. AmaLgam+: Composing Rich Information Sources for Accurate Bug Localization. *Journal of Software: Evolution and Process* 28, 10 (2016), 921–942.
[23] Andreas Zeller. 1999. Yesterday, My Program Worked. Today, It Does Not. Why?. In *Proc. of ESEC/FSE'99*. 253–267.
[24] Chenguang Zhu, Yi Li, Julia Rubin, and Marsha Chechik. 2017. A Dataset for Dynamic Discovery of Semantic Changes in Version Controlled Software Histories. In *Proc. of MSR'17*. 523–526.
[25] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. 2004. Mining Version Histories to Guide Software Changes. In *Proc. of ICSE'04*. 563–572.