Effectively Prioritizing Tests in Development Environment

Amitabh Srivastava & Jay Thiagarajan

Cyntrica Eaton
November 26, 2002

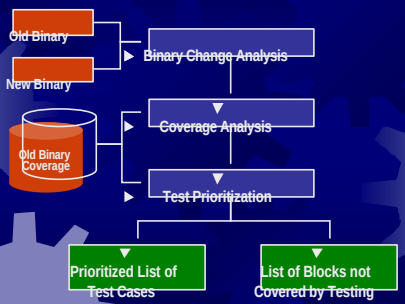
Motivation

- After initial development, software will frequently change.
- Change management is critical to maintaining software utility.
- Small changes in one part of the program may have subtle, undesired effects in other seemingly unrelated parts of the program.

Key Approach

- Effectively Prioritizing Tests…..
  - Run the right test at the right time
  - Focusing testing efforts on parts of the program affected by change.
  - New defects probably result of new modifications
- ….in Development Environment
  - Technique must be fast, useful, and easily integrated into the development process.
  - Estimate program change based on comparisons of binary representations of code.

Echelon: Basic Idea



Remaining Presentation

- Features of Echelon
- Test Prioritization Sequence
- Empirical Evaluation

Features of Echelon

- Prioritizing Tests
- Calculating Program Change
- Utilizing Program Binaries
- Providing Test Coverage Information
- General Practicality

Prioritizing Tests

- Prioritizes tests into an ordered sequence based on program change.
  - Does not permanently discard tests like minimization techniques presented in [4][13][19][30].
  - Allows use of a non-precise algorithm that works well in practice.

Calculating Program Change

- Computes changes between programs at a very fine granularity using an accurate binary matching algorithm.
  - The paper points out other works in test selection [1][3][4][20][28] and test prioritization [6][8][17][31] that also use program change as a guiding factor.
  - Cites how differences in techniques affect the estimation of program change.

Calculating Program Change

- Source code differencing
- Data and control flow analysis
- Coarse-grained code entities

Calculating Program Change

- Source Code Differencing [6][8][28]
  - Simple and fast
  - Can be accomplished with commonly available tools like “diff” in Unix
  - Erroneously marks a procedure as changed even if variables were only renamed.
  - Fails to determine the set of affected procedures when header files that define macros and methods have been modified.

Calculating Program Change

- Coarse-grained techniques [5]
  - Uses functions, global variables, type definitions, etc. to identify which parts of the program might be affected by the changes.
  - According to [11], estimations of program change based on coarse-grained entities [5] may select more tests than statement or control flow based techniques.

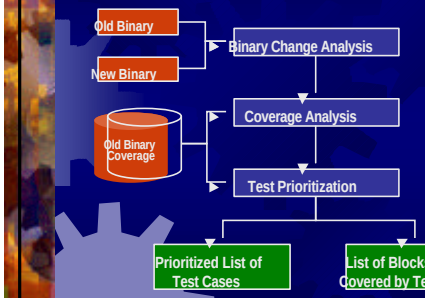
Calculating Program Change

- Data and control flow analysis [1][20]
  - Difficult in a language such as C++/C which contains pointers, casts, and aliasing.
  - Flow analysis is expensive in large commercial systems [9][21] and should not be used unless the techniques will be helpful for other analyses[9].

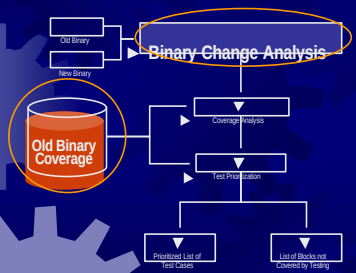
Utilizing Program Binaries

- Working at the binary level has advantages over working at the source code level.
  - Binary modification eliminates the recompilation step for collecting coverage
  - Easier to integrate into the build process in production environments.
  - Once available in binary form, all header file changes have been propagated to the affected procedures in the given program.
  - Simplifies the process for determining program changes.

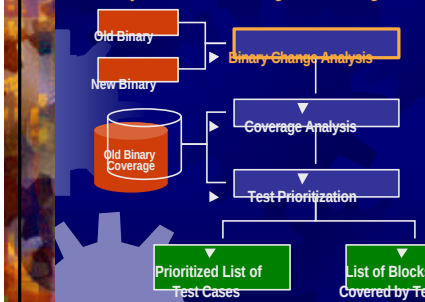
Test Prioritization Sequence



Test Prioritization Sequence



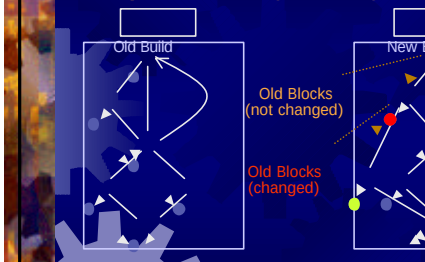
Step 1: Binary Analysis



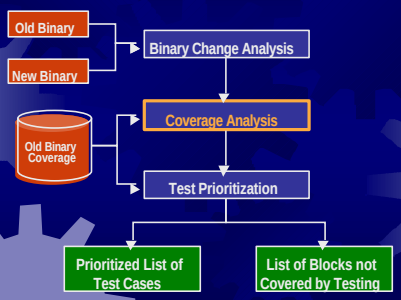
Step 1: Binary Analysis

- Uses BMAT to find a matching block in the old binary for each new block in the new binary.
  - Unmatched blocks are labeled as new blocks.
  - Blocks with matches are further compared:
    - Identical blocks are labeled as old blocks
    - Otherwise marked as old-modified blocks.

Step 1: Binary Analysis



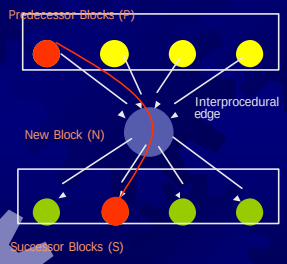
Step 2: Coverage Analysis



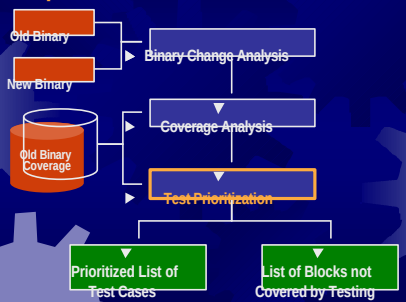
Step 2: Coverage Analysis

- Determine which impacted blocks in the new version are likely to be covered by an existing test.
  - Old modified blocks
    - Check to see if the test covered the matching block in the old binary using the cover information of the old binary.
  - New blocks
    - A test may cover a new block if it covers at least one of its immediate predecessor blocks and at least one of its immediate successor blocks, skipping in both cases, any intermediate new blocks.

Step 2: Coverage Analysis



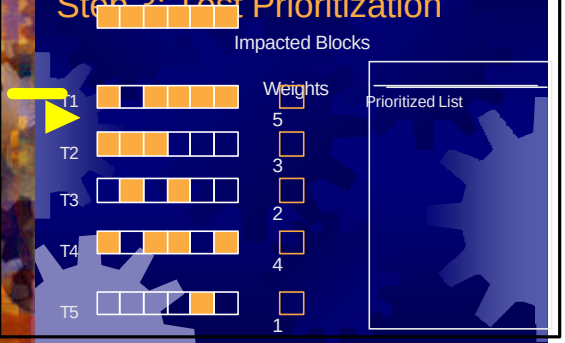
Step 3: Test Prioritization

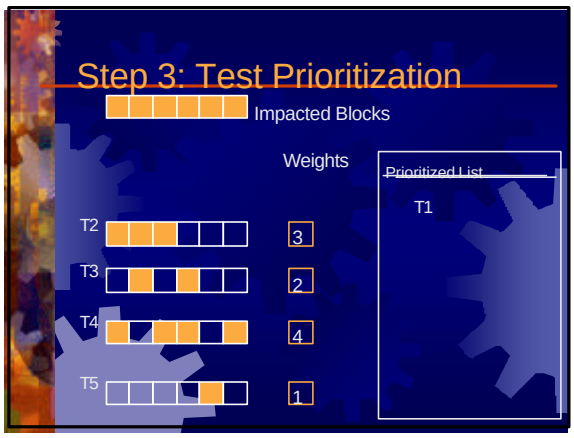


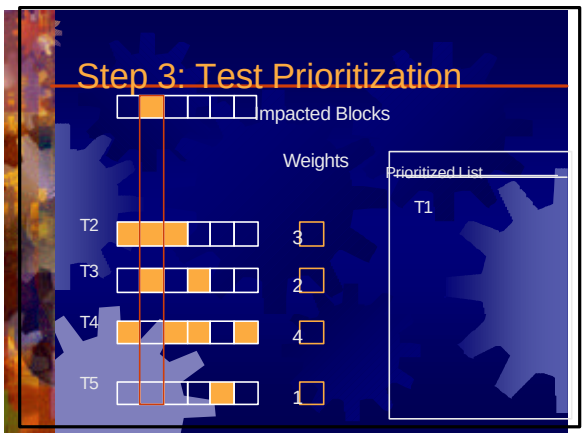
Step 3: Test Prioritization

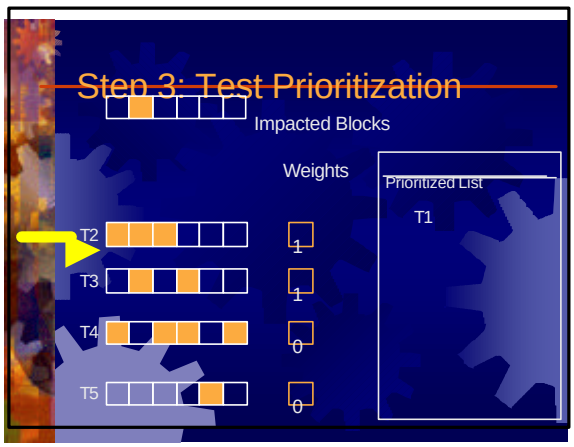
- By now, Echelon has predicted the set of impacted blocks that will be covered by each test.
- Uses the impacted block set for each test as a basis for prioritization.
- Iteratively finds a short sequence of tests which cover the maximum amount of impacted blocks.

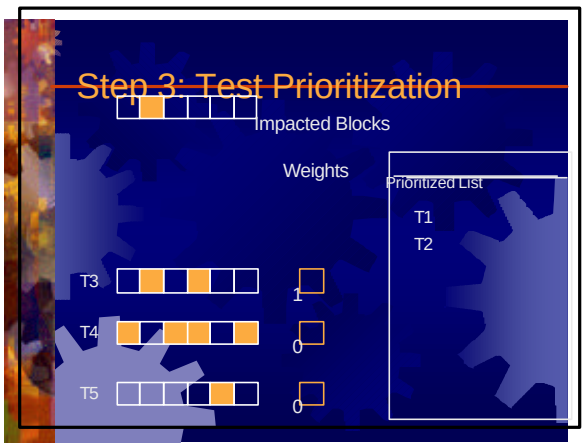
Step 3: Test Prioritization

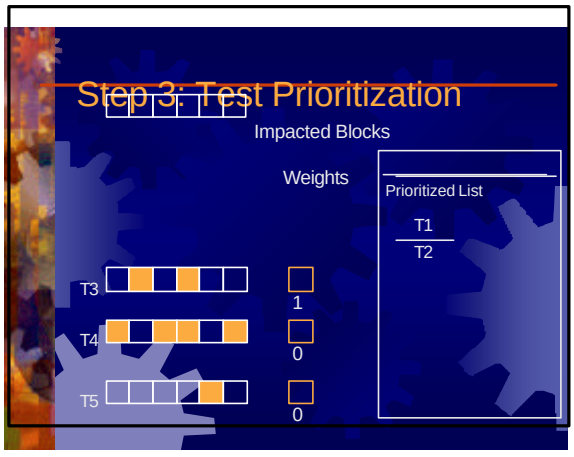


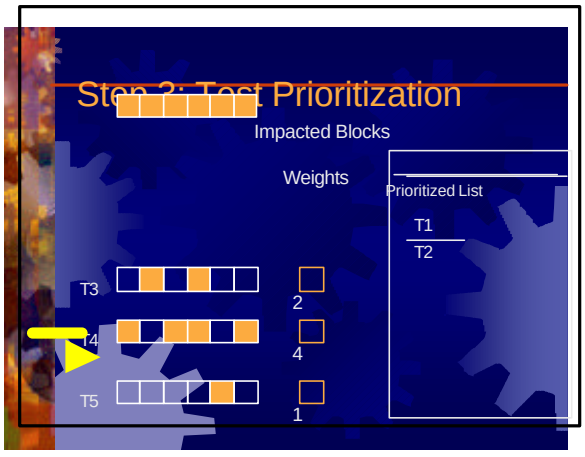

Step 3: Test Prioritization
Impacted Blocks
Weights
Prioritized List
T1
T2
T3
T4
T5
3
2
4
1

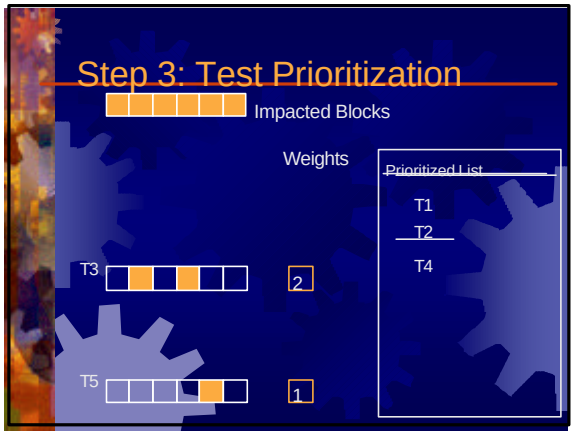

Step 3: Test Prioritization
Impacted Blocks
Weights
Prioritized List
T1
T2
T3
T4
T5
3
2
4
1

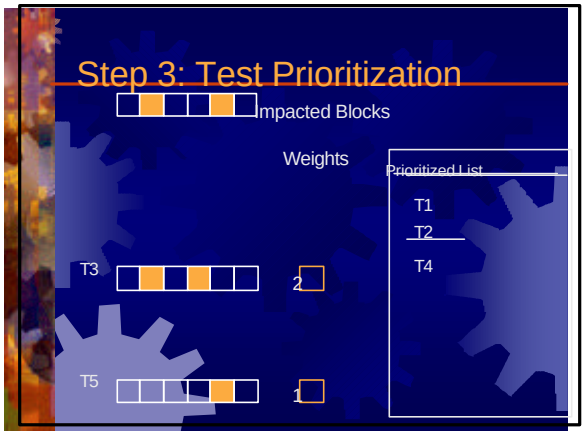

Step 3: Test Prioritization
Impacted Blocks
Weights
Prioritized List
T1
T2
T3
T4
T5
1
1
0
0

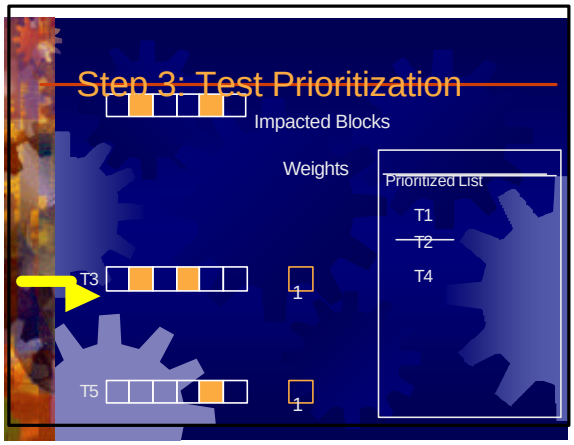

Step 3: Test Prioritization
Impacted Blocks
Weights
Prioritized List
T1
T2
T3
T4
T5
1
0
0

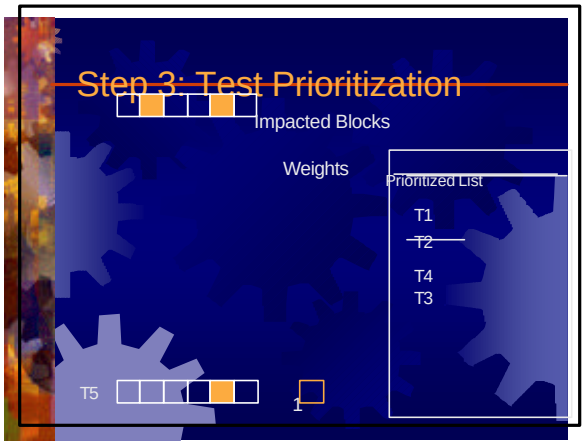

Step 3: Test Prioritization
Impacted Blocks
Weights
Prioritized List
T1
T2
T3
T4
T5
1
0
0

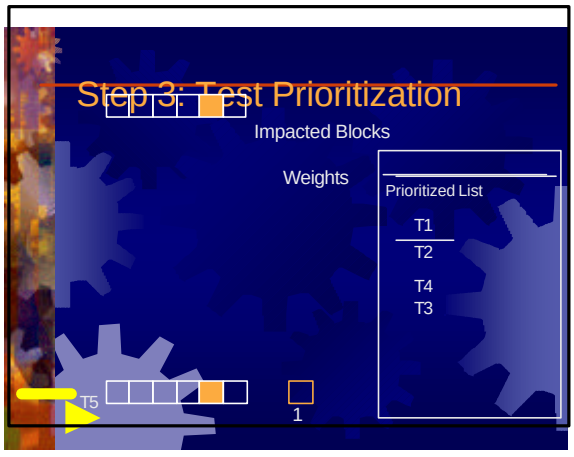

Step 3: Test Prioritization
Impacted Blocks
Weights
Prioritized List
T1
T2
T3
T4
T5
2
4
1

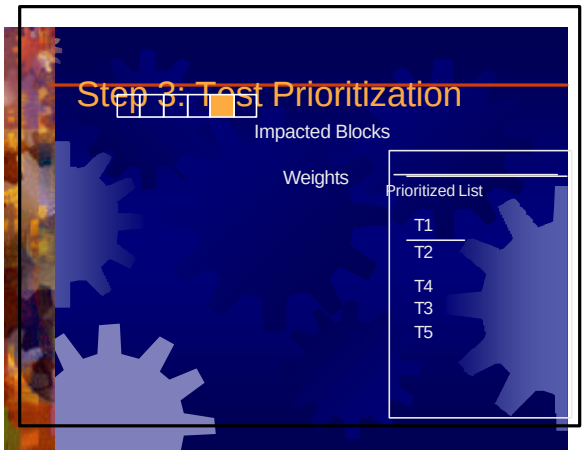

Step 3: Test Prioritization
Impacted Blocks
Weights
Prioritized List
T1
T2
T4
T3
2
T5
1


Step 3: Test Prioritization
Impacted Blocks
Weights
Prioritized List
T1
T2
T4
T3
2
T5
1


Step 3: Test Prioritization
Impacted Blocks
Weights
Prioritized List
T1
T2
T4
T3
1
T5
1


Step 3: Test Prioritization
Impacted Blocks
Weights
Prioritized List
T1
T2
T4
T3
T5
1


Step 3: Test Prioritization
Impacted Blocks
Weights
Prioritized List
T1
T2
T4
T3
T5
1


Step 3: Test Prioritization
Impacted Blocks
Weights
Prioritized List
T1
T2
T4
T3
T5

Step 3: Test Prioritization

Impacted Blocks

Weights

Prioritized List

T1
T2
} Sequence #1

T4
T3
T5
} Sequence #2

Empirical Evaluation

- Performance
  - Test sequence characteristics
  - Predicted blocked coverage accuracy
- Effectiveness
  - Early detection of defects when tests run in prescribed, prioritized order.

Performance of Echelon

	Version 1	Version2
Date	December 2000	January 2001
Functions	31,020	31,026
Blocks	668,068	668,274
File Size (bytes)	8,880,128	8,880,128
No. of Tests	3128	3128

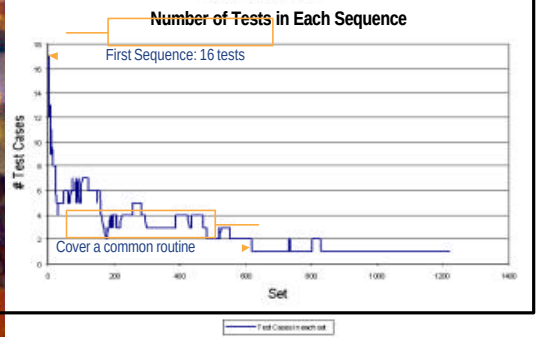
Performance of Echelon

Impacted Blocks	New Old Total	220 158 378
Impacted Blocks Covered	176 Blocks	
Tests in first seq.	16 Tests	
Number of sequences	1,225	
Time taken by Echelon	210 seconds	

Performance Analysis

- How many sequences of tests were formed?
- How many tests are in each sequence?
- How accurate is Echelon?

Performance Analysis

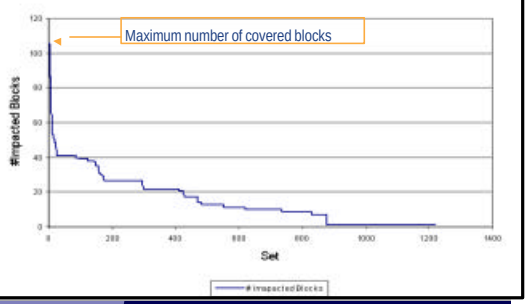


Performance of Echelon

Impacted Blocks	New	220
	Old	158
	Total	378
Impacted Blocks Covered	176 Blocks	
Tests in first seq.	16 Tests	
Number of sequences	1,225	
Time taken by Echelon	210 seconds	

Performance Analysis

Number of Impacted Blocks in Each Sequence

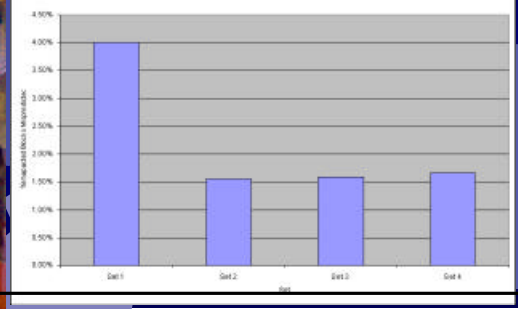


Performance Analysis

- Accuracy
  - How many blocks that were predicted to be covered by a test were not covered?
  - How many blocks that were expected to remain uncovered were actually uncovered?

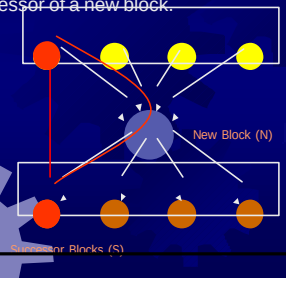
Performance Analysis

Incorrectly Predicted False Positives



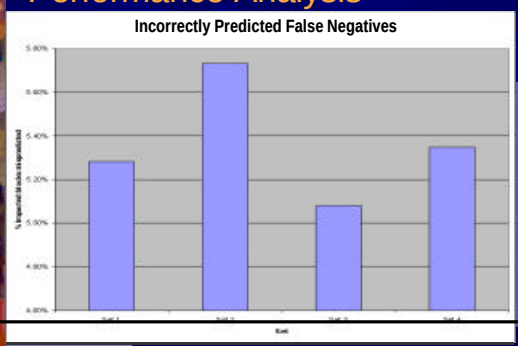
Performance Analysis

False positives can be explained by instances where there was also a direct path from the predecessor to the successor of a new block.



Performance Analysis

Incorrectly Predicted False Negatives



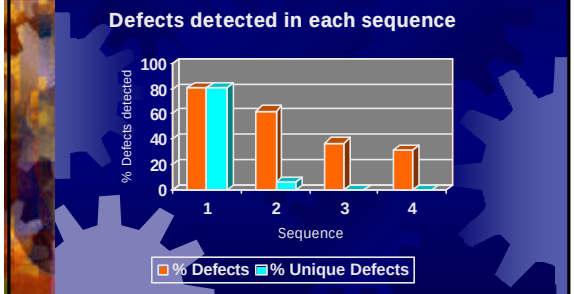
Performance Analysis

False negatives can be attributed to instances when new blocks are inserted at the head of an indirectly called procedure. Because of the manner in which the procedure is called, no predecessor of these new blocks were visible in the graph.

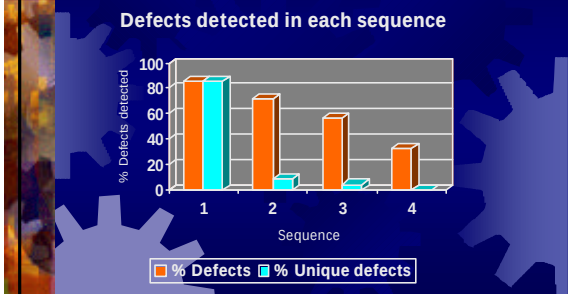
System Effectiveness

- How early can defects be detected if the tests are run in the prescribed order?
- Method:
  - Obtained a binary with a known amount of defects
  - Obtained a prioritized list of tests generated by Echelon
  - Ran tests in the prescribed order
  - Documented the number of unique defects detected for each sequence of tests

System Effectiveness



System Effectiveness



Main Conclusion

The use of binary matching to determine inter-version program changes and influence regression test prioritization is effective in large-scale production environments.

Cited References

[1] T. Ball, "On the Limit of Control Flow Analysis for Regression Test Selection". Proc. ACM Int'l Symposium. Software Testing and Analysis, pp. 134-142, Mar. 1998.

[3] D. Binkley, "Semantics guided Regression Test Cost Reduction", IEEE Trans. Software Eng., vol. 23, no. 8, pp. 498-516, Aug. 1997.

[4] T. Y. Chen and M. F. Lau, "Dividing Strategies for the Optimizaton of a Test Suite", Information Processing Letters, vol. 60, no. 3, pp. 135-141, Mar. 1996.

[5] Y. F. Chen, D. S. Rosenblum, and K. P. Vo, "TestTube: A System for Selective Regression Testing," Proc. 16th Int'l Conf. Software Eng., pp. 211-222, May 1994.

[6] S. Elbaum, .A. Malishevsky and G. Rothermel, "Test case prioritization: A family of empirical studies", IEEE Trans. Software Engg. , vol. 28, no. 2, pp. 159-182, Feb. 2002.

[8] S. Elbaum, .A. Malishevsky and G. Rothermel, "Prioritizing test cases for regression testing", Proc. Int'l Symp. Software Testing and Analysis, pp. 102-112, Aug. 2000.

[9] T. L. Graves, M. J. Harrold, J-M. Kim, A. Porter and G. Rothermel, "An empirical study of regression test selection techniques", 20th Int'l Conference on Software Engineering, Apr. 1998.

[11] M. J. Harrold, "Testing Evolving Software", Journal of Systems and Software, vol. 47, no. 2-3, pp. 173-181, Jul. 1999.

Cited References

[13] M. J. Harrold, R. Gupta and M. L. Soffa, "A Methodology for Controlling the Size of a Test Suite", ACM Trans. Software Eng. And Methodology, vol. 2, no. 3, pp. 270-285, July 1993.

[17] G. Rothermel, R. H. Untch and M. J. Harrold, "Prioritizing Test Cases For Regression Testing", IEEE trans. On Software Engineering, vol. 27, no. 10, Oct. 2001

[19] G. Rothermel, M. J. Harrold, J. Ostrin and C. Hong, "An Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test Suites", Proc. Int'l Conf. Software Maintenance, pp. 34-43, Nov. 1998.

[20] G. Rothermel and M. J. Harrold, "A Safe, Efficient Regression Test Selection Technique", ACM Trans. Software Eng. And Methodology, vol. 6, no. 2, pp. 173-210, Apr. 1997.

[28] F. Vokolos and P. Frankl, "Pythia: a regression test selection tool based on text differencing", Int'l conference on reliability, Quality and Safety of SoftwareIntensive Systems, May 1997.

[30] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of Test Set Minimization on Fault Detection Effectiveness", Software-Practice and Experience, vol. 28. no. 4, pp. 347-369, Apr. 1998.

[31] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal, "A Study of Effective Regression Testing in Practice", Proc. Eighth Int'l Symposium Software Reliability Eng., pp. 230-238, Nov. 1997.

Effectively Prioritizing Tests in Development Environment

Amitabh Srivastava & Jay Thiagarajan

Cyntrica Eaton

November 26, 2002