

Inferring Developer Activities by Analyzing Successive Versions of Source Code

Jaymie Strecker

December 17, 2004

1 Introduction

As a software program is developed, the developers make incremental changes to the source code, producing over time a succession of versions of the code. In many cases, a version control system or some other instrumentation records the changes made from version to version. Is it possible to use such readily available information about source code changes to extrapolate fine-grained information about developer activities?

The High Productivity Computer Systems (HPCS) Development Time Working Group has collected version data from numerous classroom experiments which tracked students' progress on assignments in HPC programming courses (Carver et al., 2004; Asgari et al., 2004). Since HPCS seeks to understand and improve the development process with respect to effort and quality, it is valuable to determine why a developer made a particular change to the source code. Each change costs the developer some effort, and more defects require more changes to fix them.

Currently, HPCS collects data about developer activities through instrumented compilers. In the experiments, subjects use an instrumented compiler, which is a wrapper around their usual compiler. At each attempt to compile, the instrumented compiler asks the subject to select one of several options to describe the reason for any changes made since the previous attempt (Carver et al., 2004; Asgari et al., 2004).

The approach presented here for inferring developer activities improves upon instrumented compilers in several ways. Our approach is not only transparent to the subject, but it can also be applied *ex post facto*, assuming a version control system or other instrumentation automatically tracks the subject's source code development. Our approach infers developer activity at a finer granularity than is practical for most experiments, since it would rarely be practical to ask subjects to classify every change they made. Finally, our approach is more objective and consistent than self-reported data from subjects.

In this paper, we present a model for source code changes that extends existing change models with a new dimension: developer activity. Then, we present heuristic algorithms which guess the developer activity that resulted in a given source code change, based on the source code of the versions related to the change. Finally, we evaluate the performance of a change classification tool that implements the heuristic algorithms.

2 Related Work

2.1 Counting and Classifying Changes

Changes made during software development have been studied at various levels of detail. Weiss and Basili (Weiss and Basili, 1985) developed a set of change classes based on their observations of several projects at NASA’s Software Engineering Laboratory. Encompassing the entire development process, not just source code development, their change classes included requirements modifications, planned enhancements, improvements of clarity, improvements of services, and optimizations. They found that the distribution of changes over these classes differed among projects.

At a finer level of detail, Dunsmore and Gannon (Dunsmore and Gannon, 1978) present guidelines for counting changes in source code. Their purpose for counting changes is to verify their hypothesis that the number of changes in a program correlates with indicators of the program’s complexity, such as the number of error occurrences. Most of their change counting guidelines agree with the approach used in this paper, but unlike Dunsmore and Gannon, we include changes to debugging statements and documentation.

Several projects have focused on maintenance changes as recorded in the repositories of version control systems (Hassan et al., 2004). German (German, 2004) performs data mining on CVS repositories, ChangeLogs, Bugzilla, and mailing lists to discover “software trails”, which he defines as “information left behind by the contributors to the development process, usually in the form of logs”. He counts changes at the level of Modification Requests, which are groups of files that are modified together by a CVS commit or equivalent command.

Mockus and Votta (Mockus and Votta, 2000) rely on the textual descriptions of changes, as recorded in a version control system repository, to classify the changes. Like German, they examine changes at the level of Modification Requests. They describe the algorithm by which they classify changes based on keywords in their textual description. After verifying that their change classification scheme is consistent with developers’ self-reported classification, they apply their classification scheme to two software products made by the same company. For these products, each change class has a characteristic size, time interval, and difficulty for changes.

Purushothaman and Perry (Purushothaman and Perry, 2004) characterize changes not only by their size, but more specifically by whether their size is one line. One of their goals is to test the commonly held belief that “one-line changes are erroneous 50 percent of the time”, which they find untrue in the context of the multi-million line software product they study. Using Mockus and Votta’s change classification system to ascertain the purpose of each change, they observe a number of relationships among the size, purpose, and error-proneness of changes. For example, they report that “nearly 40 percent of changes that were made to fix defects introduced one or more defects in the software”.

2.2 Key Differences from Previous Work

Mockus and Votta (Mockus and Votta, 2000) have hypothesized that source code alone does not contain enough information to classify changes by their purpose; a textual description is essential. One of the goals of this study is to refute this hypothesis within certain contexts. In the context of this study, the source code data analyzed is more detailed than data from many software repositories, since our data include, for example, versions that fail to compile and versions with temporary debugging code. In this study, we use a bottom-up approach. Several heuristics pick out some common types of changes in the code, leaving the remaining changes for the user of our approach to classify on her own.

3 Change Models

3.1 Definition of a Change

At a large scale, researchers have defined a change to software as a change request within an organization (Weiss and Basili, 1985) or a Modification Request within a software repository (German, 2004; Mockus and Votta, 2000). At a small scale, Dunsmore and Gannon (Dunsmore and Gannon, 1978) state that “one program change should be concerned with the contiguous set of concrete statements that represent a single abstract instruction”. Since changes by this definition are difficult to identify algorithmically, in this study we approximate this definition by considering any contiguous set of modified lines (where the modification includes characters other than whitespace) to be a single change. In other words, the number of changes between two source code versions is exactly the number of results returned by the `diff` utility with flags `-bB` (MacKenzie et al., 2000).

In the following example, there are three changes from Version 1 to Version 2: an addition at line 1 of Version 2, a deletion at line 4 of Version 1, and a replacement from line 6 of Version 1 to lines 6 and 7 of Version 2.

```
--- Version 1 ---
main() {
    int a, b;
    int c;
    // looping
    for (b = 0; b < 10; b++)
        a = b + 1;
    c = a;
}

--- Version 2 ---
#include <stdio.h>
main() {
    int a, b;
    int c;
    for (b = 0; b < 10; b++)
        a = b + 2;
        printf("a=%d\n", a);
    c = a;
}
```

3.2 Existing Change Models

In the literature, most change classification schemes have been concerned with changes made during the maintenance phase (Hassan et al., 2004). Many proposed classifications have been closely related to the set of maintenance change types that includes adaptive, corrective, and perfective changes (Mockus and

Table 1: Dimensions of the change model and their values.

Dimension	Values
developer activity	program, debug, or optimize
purpose	adaptive, corrective, or perfective
modification type	addition, deletion, or replacement
size	number of lines changed
location	any functions the change lies within

Votta, 2000; Purushothaman and Perry, 2004; Briand and Basili, 1992). Adaptive changes are those that add functionality to the software product. Corrective changes are made to correct defects. Perfective changes modify the code in order to make it easier to change in the future. Changes are made for these purposes during the development process as well as the maintenance process.

In addition to the adaptive, corrective, or perfective purpose of a change, other dimensions of changes have been used to classify them. These include the type of modification (addition, deletion, or replacement), the size of the change, and the location of the change (Purushothaman and Perry, 2004; Briand and Basili, 1992; Mockus and Votta, 2000; Weiss and Basili, 1985).

3.3 A Change Model with Developer Activity

Since the purpose of this study is to infer a developer’s activities from the successive versions of source code she produces, we would like to classify source code changes with respect to the developer activity that produced them, in addition to other dimensions used in existing change models. This extended change model is summarized in Table 1.

For the set of developer activities in our model, we use part of the set proposed and defined by Smith (Smith, 2004). During the development process, a developer performs all or most of the following activities at various times:

- **Formulate** - “Formulate an algorithmic approach”.
- **Program** - “Create or incrementally augment the program and its testing infrastructure”.
- **Compile** - “Compile and link the program developed so far”.
- **Test** - “Test the program, observing its behavior”.
- **Debug** - “Diagnose and fix erroneous behavior”.
- **Run** - “Run the program on the real input data”.
- **Optimize** - “Improve program performance”.

Some of these activities, like Formulate and Run, typically leave no trace in the source code. In this study, we are only concerned with those activities that do.

Table 2: For the developer activities in the change model, some lower-level, more concrete actions that comprise those activities are shown. The purpose of a change is determined by the low-level activity.

Developer activity	Low-level activity	Purpose
Program		
	Add functionality	adaptive
	Correct compile-time errors	corrective
	Modify comments	perfective
	Refactor	perfective
Debug		
	Correct run-time errors	corrective
	Modify debugging code	other
	Comment/uncomment executable statements	other
Optimize		perfective

4 A Heuristic Algorithm for Classifying Changes

When classifying changes based on the change model in Table 1, the type of modification, size of the change, and location of the change are evident from the source code. Inferring the developer activity is the only hurdle to overcome, since the purpose of the change is directly related to the developer activity, as Table 2 shows.

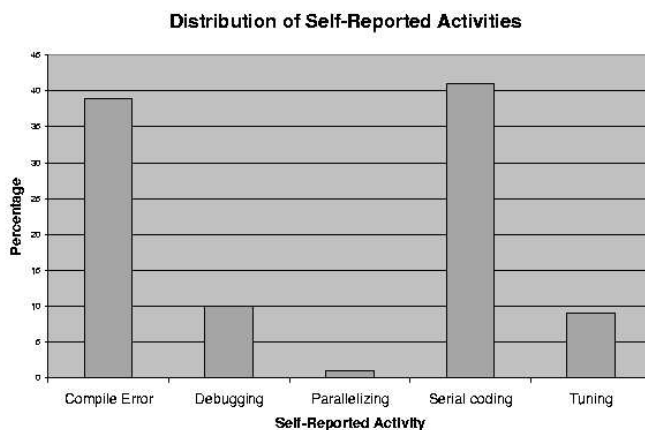
Our approach in classifying changes to source code is to check the changes for some patterns that are indicative of certain developer activities. This approach is inspired by the FindBugs tool, which searches for bug patterns within single versions of source code (Hovemeyer and Pugh, 2004). In Table 2, some commonly performed, low-level activities are listed alongside the higher-level developer activities used in the change model. In the change classification algorithm, we search for traces of some of these low-level activities in the source code versions, mainly by inexpensive techniques like pattern matching.

In order to classify a given change between consecutive versions A and B of a piece of software, we apply the heuristics listed below. The change is classified by the first heuristic that recognizes it, so the order in which the heuristics are applied matters. For example, a change that comments out an executable statement is recognized and classified by Heuristic 3, even though Heuristic 4 would have also recognized it.

1. **Add Functionality I.** If Version A is an empty file, that is, if Version B reflects any changes made from the start of coding to the saving of the first version, then the developer activity is **Program**.
2. **Correct Compile-Time Errors.** If Version A does not compile, then the developer activity is **Program**.

3. **Comment/Uncomment Executable Statements.** If some executable statement exists in both Version A and Version B, and the statement is inside a comment in one version but not the other, then the developer activity is **Debug**.
4. **Modify Comments.** If text inside comments is added, deleted, or replaced, then the developer activity is **Program**.
5. **Modify Debugging Code.** If the change includes a print statement which does not appear in the final (or last available) version of the source code, then the developer activity is **Debug**.
6. **Add Functionality II.** If the change is an addition, then the developer activity is **Program**.

Any remaining changes are unclassified.



5 Implementation and Analysis

To enable rapid data analysis, we implemented these heuristics in an automated change classification tool. The implementation is straightforward; it follows directly from the change model in Table 1 and the definitions of the heuristics. Given a directory that contains successive versions of a source code file, the tool produces a comma-delimited output file which contains the classification for each change found in the input. To locate changes, the tool uses the `diff` utility. To determine if a source version compiles, the tool can use any compiler that the user specifies.

We performed a pilot study of the tool using data collected in the field to evaluate the heuristics and identify areas for improvement. This data was collected by instrumented compilers during a classroom experiment. In the experiment, ten students in a graduate-level parallel programming course worked individually to write serial C/C++ and parallel programs for three class assignments. Most of the students' programs were comprised of a single file. When a student compiled her source code, the instrumented compiler required the student to select one of the developer activities listed in Table 3. At each attempt to compile, the instrumentation recorded the student's self-reported activity, the current state (version) of the source code, and whether the source code successfully compiled.

In this study, we analyzed only the serial versions of the students' assignments, using the compiler `g++`. Changes for which the subject reported one of the starred activities in Table 3 were not considered in the analysis of the tool's performance. Most of the starred activities, by definition, leave the source code for the assignment unchanged. Figure 5 shows the distribution of self-reported activities for the data analyzed.

We evaluate the performance of the change classification tool on these data with respect to two ideals. The first ideal is accuracy, which means that any

Table 3: Subjects were given the activity options and definitions shown here. In order for the tool to infer the developer activity of a change correctly, one of the heuristics listed for the self-reported activity must recognize the change. Starred self-reported activities were not considered in the evaluation of the tool.

Self-reported activity	Definition	Heuristics
*Thinking	Work not on the computer including thinking about a solution to the problem, planning/designing your implementation	
Serial Coding	Creating a serial version of your solution	Add functionality I, Add functionality II, Modify comments
Parallelizing the Code	Modifying your serial program to work on more than one processor	Add functionality I, Add functionality II, Modify comments
*Testing Code	Verifying that the answer produced by your code is correct, timing your code for performance evaluation, etc.	
Debugging	Identifying the reason for a failure and fixing it	Correct compile-time errors, Correct runtime errors, Modify debugging code, Comment/uncomment executable statements
Tuning Code	Modifying the code to improve performance	
*Experimenting with the Environment	Work not done specifically on one of the assignments (e.g. writing “hello world”, learning how to use the job scheduler)	
*Other	Any other tasks (please specify)	
Compile-Time Error	Reported automatically by the instrumented compiler	Correct compile-time errors

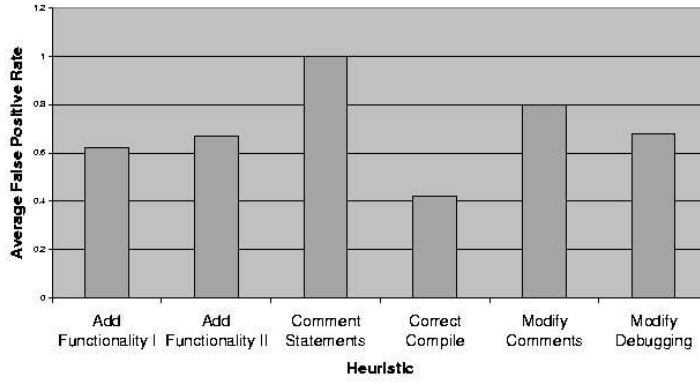
changes the technique classifies, it classifies correctly. The classification of a change is considered to be correct if the heuristic that recognizes the change is listed in Table 3 alongside the self-reported activity of the change. Since subjects reported their activities only when they invoked the compiler, the self-reported activity is the same for all changes made between two successive versions of the source code. The second ideal, precision, means that the technique leaves no changes unclassified. We use the following metrics to perform this evaluation:

- **Number of false positives per heuristic per subject.** For heuristic X and subject Y, this is the number of incorrectly classified changes recognized by X for Y.
- **False positive rate per heuristic per subject.** For heuristic X and subject Y, this is the number of false positives for X, divided by the total number of changes recognized by X, for Y.
- **Average false positive rate per heuristic.** For heuristic X, this is the false positive rate for X, averaged across all subjects. See Figure 5 and Figure 5 for results.
- **Percentage of changes unclassified per subject.** For subject Y, this is the percentage of changes made by Y that the tool is unable to classify. See Figure 5 for results.
- **Percentage of changes unclassified overall.** This is the percentage of all changes that the tool is unable to classify. See Figure 5 for results.
- **Percentage of unclassified changes per self-reported activity.** For self-reported activity Z, this is the number of changes reported as Z that the tool is unable to classify, divided by the total number of unclassified changes. See Figure 5 for results.

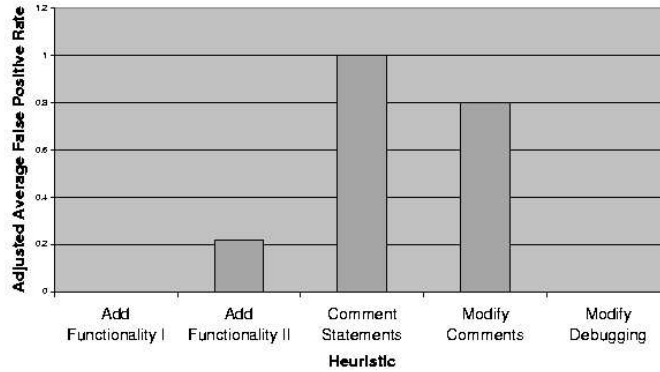
Because a discrepancy was discovered between the instrumented compiler's reporting of compile-time errors and the scoring of the tool's classification as correct or incorrect, the data were reanalyzed after removing changes in which the instrumented compiler reported a Compile-Time Error or the tool classified the change with the Correct Compile-Time Errors heuristic. Ninety percent of the changes were removed from the data, and of these, 93% had been recognized by the Correct Compile-Time Error heuristic. Adjusted average false positive rates for the remaining heuristics are shown in Figure 5.

Figure 5 compares the run time of the components of the tool with the entire run time of the tool. Figure 5 compares the run time of the tool on a Sun machine with a 502 MHz sparcv9 processor to the run time on a Linux PC with a 2 GHz processor.

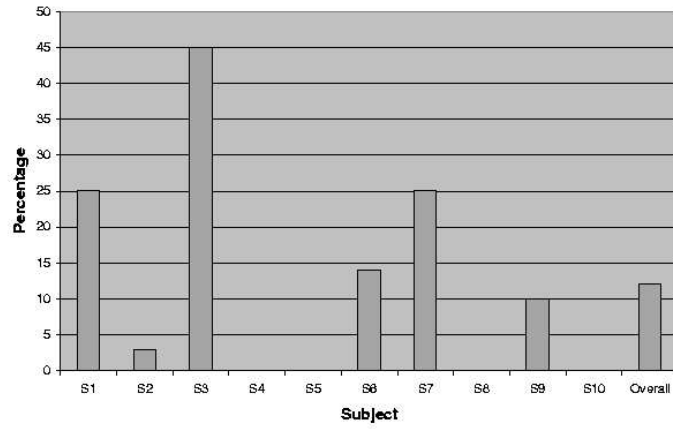
Average False Positive Rate



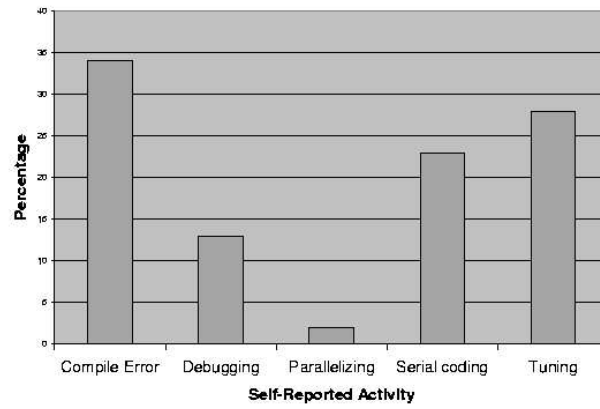
Adjusted Average False Positive Rate

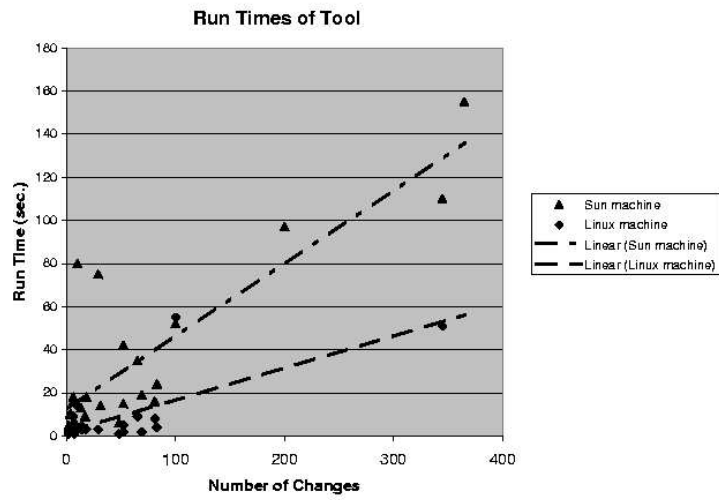
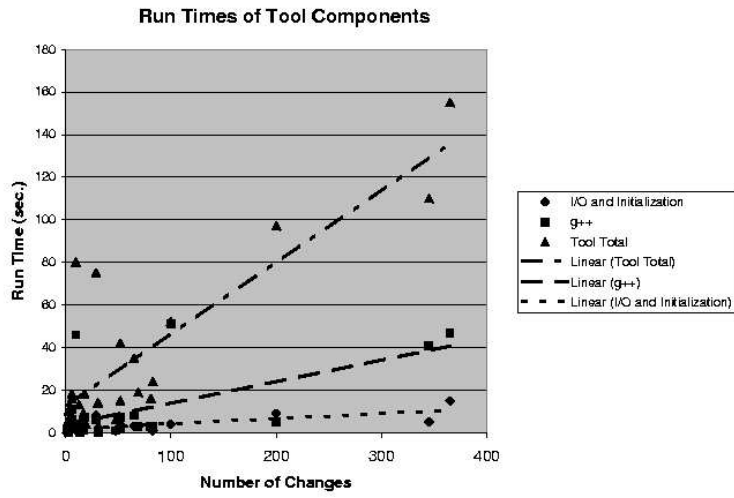


Percentage of Changes Unclassified



Distribution of Unclassified Changes





6 Conclusions

That 13% of the changes found, mostly self-reported as Testing, were discarded indicates that the self-reported activity data may not be very reflective of the changes in the source code. For example, if a subject spends 20 minutes in the Thinking activity, followed by 5 minutes in the Serial coding activity, the subject will likely select the Thinking activity when asked to describe her activities of the past 25 minutes. Also, subjects may have forgotten the definitions of the activities or interpreted them differently than the researchers.

A significant proportion of the false positives initially discovered were a product of the discrepancy in the scoring system involving compiler errors. Assuming that the compiler specified by the user of the tool is the same as the compiler used by the subjects, the Correct compile-time errors heuristic is expected to have perfect accuracy. In the reanalysis of the data, the false positive rates for the heuristics Add Functionality I, Add Functionality II, and Modify Comments fell to reasonable levels.

The heuristics Comment/Uncomment Executable statements and Modify Debugging Code had high false positive rates in both the initial and the adjusted data. For Comment/Uncomment Executable Statements, all of the changes recognized by this heuristic came from a single subject. Although the subject reported Serial or Tuning as the related activity, a manual analysis of the changes shows that most of them actually consisted of a single print statement (apparently used for debugging) being commented or uncommented. The subject may not have selected the Debugging activity because the effort spent on these changes was overshadowed by that spent in other activities, or, since none of the changes in this subject's serial source files were reported as Debugging, perhaps the subject had ignored that activity category for some reason.

The high false positive rate of the Modify Debugging Code heuristic may be explained in much the same way. Subjects selected the Serial Coding or Tuning activity for many changes to print statements like the following, quoted from one subject's source file (with variable names changed):

```
fprintf(stdout, "variables:%d_%d_%s\n", a, b, c);
```

In the source code, this statement prints the value of variables as they are read from a file.

Of the unclassified changes, a third were reported by the instrumented compiler as having compile-time errors. All of these changes would have been classified if not for the discrepancy in the scoring system involving compiler errors. For the unclassified changes, the self-reported activities were Serial Coding, Tuning, and Debugging.

7 Future Work

New heuristics need to be added to infer the developer activities of changes more accurately and precisely. The choice of new heuristics to add should be driven by the interestingness to the stakeholders (researchers and educators) of the developer activities that the heuristics detect. In particular, since HPCS studies the types of mistakes developers make using various programming paradigms and languages, the heuristics we create will be defect-driven and sometimes language-specific.

To increase accuracy, the definition of a change used in our model may need to be refined so that a change encompasses a set of consecutive lines *all changed due to the same developer activity*. If several consecutive lines are changed, but the first line is changed to add a comment and the rest to add functionality, then, by the refined definition, these consecutive lines would be split into two changes. However, since about 60% encompassed just one line, the need to refine the change definition is not as urgent as we had initially believed.

Only the developer of a source code knows for sure why she makes a certain change to the source code, and possibly only for a short time after the change is made. Therefore, self-reported activity data from developers is essential to verify that our change classification system is useful. However, since it is not known how accurate self-reported activity data from instrumented compilers actually are, a more controlled data collection method should be used in the future when evaluating the change classification system.

References

- Asgari, S., Basili, V., Carver, J., Hochstein, L., Hollingsworth, J., Shull, F., and Zelkowitz, M. (2004). Challenges in measuring hpcs learner productivity in an age of ubiquitous computing. In *Workshop on Software Engineering and High Performance Computing Applications (co-located with ICSE)*.
- Briand, L. C. and Basili, V. R. (1992). Changes during the maintenance process. In *Conference on Software Maintenance*.
- Carver, J., Asgari, S., Basili, V., Hochstein, L., Hollingsworth, J., Shull, F., and Zelkowitz, M. (2004). Studying code development for high performance computing: The hpcs program. In *Workshop on Software Engineering and High Performance Computing Applications (co-located with ICSE)*.
- Dunsmore, H. E. and Gannon, J. D. (1978). Programming factors - language features that help explain programming complexity. In *ACM/CSC-ER Annual Conference, Volume 2*, pages 554–560.
- German, D. M. (2004). Mining cvs repositories, the softchange experience. In *1st International Workshop on Mining Software Repositories*, pages 17–21.
- Hassan, A. E., Holt, R. C., and Mockus, A., editors (2004). *1st International Workshop on Mining Software Repositories*.
- Hovemeyer, D. and Pugh, B. (2004). Finding bugs is easy. In *OOPSLA (Onward! Track)*.
- MacKenzie, D., Eggert, P., and Stallman, R. (2000). *Comparing and Merging Files with GNU diff and patch*. Free Software Foundation, Inc.
- Mockus, A. and Votta, L. G. (2000). Identifying reasons for software change using historic databases.
- Purushothaman, R. and Perry, D. (2004). Towards understanding the rhetoric of small changes.
- Smith, B. (2004). A timed markov process model for researcher workflow. Cray, Inc.
- Weiss, D. M. and Basili, V. R. (1985). Evaluating software development by analysis of changes: The data from the software engineering laboratory. *IEEE Transactions on Software Engineering*, pages 157–168.