Dex: A Semantic-Graph Differencing Tool for Studying Changes in Large Code Bases

Shruti Raghavan, Rosanne Rohana, David Leon, Andy Podgurski, Vinay Augustine
Electrical Engineering & Computer Science Department
Case Western Reserve University
10900 Euclid Avenue
Cleveland, Ohio
+1 216 368 6884
{sxr67, rjr11, dzl, podgurski, vja2}@ case.edu

Abstract

This paper describes an automated tool called Dex (Difference extractor) for analyzing syntactic and semantic changes in large C-language code bases. It is applied to patches obtained from a source code repository, each of which comprises the code changes made to accomplish a particular task. Dex produces summary statistics characterizing these changes for all of the patches that are analyzed. Dex applies a graph differencing algorithm to abstract semantic graphs (ASGs) representing each version. The differences are then analyzed to identify higher-level program changes. We describe the design of Dex, its potential applications, and the results of applying it to analyze bug fixes from the Apache and GCC projects. The results include detailed information about the nature and frequency of missing condition defects in these projects.

1. Introduction

The history of changes made to the code base of a software system or application embodies a wealth of information about deficiencies in the software that were discovered and addressed during its lifecycle. This sort of information is invaluable for assessing software engineering techniques that are intended to prevent, reveal, or correct such deficiencies. For example, many software testing techniques are based on assumptions about the nature and frequency of software defects. These assumptions can be checked by studying bug fixes applied in actual software projects. Moreover, doing so leverages work already done by developers. Nevertheless, such studies are seldom conducted, presumably because of the time and effort required.

The biggest obstacle to conducting rigorous empirical studies of the nature of software changes is the amount of manual labor required to analyze them, which is often substantial. Another problem is that developers' own descriptions of changes and the reasons for them are typically informal and vary in specificity from developer to developer. These problems can be addressed by providing *automated support* for analyzing the code changes recorded in *source code repositories*. Most large projects maintain such a repository with the aid of a version control system such as CVS [8]. It is typically the most complete history of changes to a project's code base that is available for study. Such repositories are often so large that manually analyzing all of the code changes recorded in them would be prohibitively expensive. Analysis of the changes recorded in source code repositories is aided by the use of *text differencing* tools such as Gnu *diff* [11]. However, because such tools do not understand program syntax and semantics, they can provide only limited help [4][14][27].

This paper describes an automated tool called *Dex* (***D****ifference* ***ex****tractor*) for analyzing *syntactic and semantic changes* in large *C*-language code bases. *Dex* is meant to be useful both to *software developers* who wish to understand the nature of software changes made in their projects and to *software engineering researchers* studying the nature of typical code changes. It is applied to *patches* extracted from a source code repository, where a patch consists of all the code changes made to accomplish a particular task such as fixing a bug.[1] *Dex* produces summary statistics characterizing these changes for all of the patches that are analyzed. Note that a single patch may affect multiple source files. *Dex* compares the original and modified version of each of these files to determine their differences and aggregates the results. To enable syntactic differences and certain semantic

[1] It is up to the user of *Dex* to gather the source files and patches to be analyzed. That is, *Dex* does not interface directly with a version control system.

Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04)
1063-6773/04 $20.00 © 2004 **IEEE**

differences to be determined, *Dex* uses *CPPX* [22][23] to create an *abstract semantic graph* (*ASG*) representation of each version (see Section 3) and then applies a *graph differencing* algorithm to the resulting pair of ASGs. This algorithm produces an *edit script* that describes how the ASG for the original version of the file can be converted into the ASG for the modified version by matching, inserting, deleting, updating, or moving nodes. This edit script is then analyzed to obtain a profile of the differences between the two files.

Dex currently collects 398 different statistics (counts) about each patch that is analyzed. These statistics are then aggregated over all patches to produce summary statistics, such as the percentage of patches that exhibit a particular type of change. It is fairly easy to modify *Dex* to collect other statistics that are defined in terms of changes to a program's ASG. We evaluated its accuracy, performance, and usefulness by using it to analyze bug fixes from two large open-source software development projects: the *Apache* HTTP server and the *GCC* compiler suite. In these experiments *Dex* showed good accuracy and acceptable performance. In addition, the statistics it produced revealed some facts about the nature of bug fixes in *GCC* and *Apache* projects that have important implications for how this software should be validated. To our knowledge, these are the first detailed statistics to be presented in the literature that characterize missing condition defects in major software products.

2. Applications of *Dex*

Dex provides a practical means to study the nature of code changes in large software projects, to justify or evaluate proposed software engineering techniques, and possibly to improve them. Perhaps the most obvious application of *Dex* is to automate the process of software *defect classification* [7] and to make it more precise, objective, and repeatable. Our original reason for creating *Dex* was to analyze bug fixes in large code bases in order to determine the kinds of execution profiling that should be used in conjunction with *observation-based testing* [9][18].[2] We are particularly interested in revealing *missing condition defects*, which involve omitted conditional code and are notoriously difficult to expose. Hence, many of the statistics *Dex* currently generates address such defects. As shown in Section 7, *Dex* provided valuable information about the nature and frequency of missing condition defects in the *Apache* and *GCC* projects. *Dex* may also prove useful in other kinds of software testing. A number of *regression testing* techniques employ textual, binary, or dependence-based differencing algorithms to enable test cases to be selected or prioritized based on whether they exercise changed code [4][24]. *Dex* could be used in these applications to obtain a more precise characterization of program changes. We believe that with appropriate modifications, *Dex* can provide information that is useful for evaluating a variety of other software engineering techniques related to software maintenance.

[2] Observation-based testing involves applying statistical and data mining techniques to profiles of software tests or captured operational executions in order to identify "suspicious" executions to be audited manually.

3. Abstract semantic graphs

An *abstract semantic graph* (*ASG*) is an abstract syntax tree (AST) with extra edges indicating certain semantic information, namely type information. We call these extra edges *non-tree edges*. They connect literals and declarations to their types and variable references to their variable declarations. Figures 1 and 2 show two ASTs; a sample of non-tree edges have been added to the AST in Figure 1. The *CPPX* tool [22] uses a modified version of *GCC* to generate ASGs from *C* source code. The modified *GCC* runs the *C*-preprocessor on the source code, parses the result, performs some semantic analysis, and generates an optimized internal tree representation. The internal representation is then translated into a *Datrix* ASG in *GXL* format [13]. The representation is a computationally equivalent program, but is not a direct translation of the original source code. This process also strips layout, spacing and comments, resolves macros to their implementation, strips unnecessary operators, converts boolean expressions to conjunctive normal form, folds literals, and expands expressions that have implicit meaning.

4. Architecture

Figure 3 depicts the architecture of *Dex* and the data flow between its components. As indicated in the figure, *CPPX* is applied to pairs of source code files (original and modified) comprising a patch, to produce files containing corresponding abstract semantic graphs in *GXL* format. The latter are input to the *DifferenceAnalyzer*, which is *Dex*'s driver module. The *DifferenceAnalyzer* parses user input parameters and initiates analysis of each pair of *GXL* files. The *GXLParser* parses the files and creates internal representations of the ASGs they describe. It passes the two ASGs to the *GraphComparator* for comparison. The *GraphComparator* uses the *CostCalculator* to determine edit distances between nodes and produces an edit script called a *NodeMatching* that classifies nodes as being matched, inserted, deleted, updated, and/or moved.

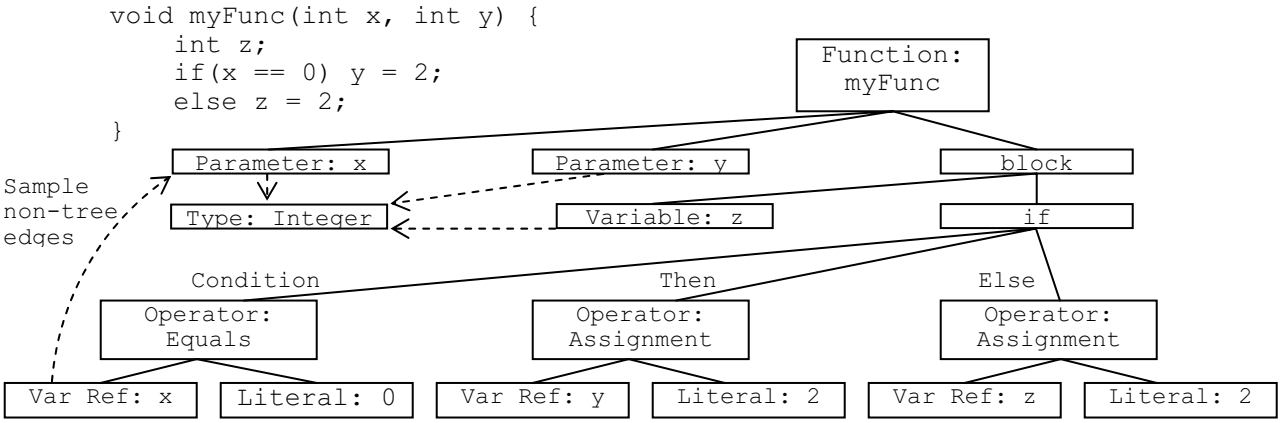

Figure 1: Simple C function and corresponding AST. Some sample non-tree edges shown.

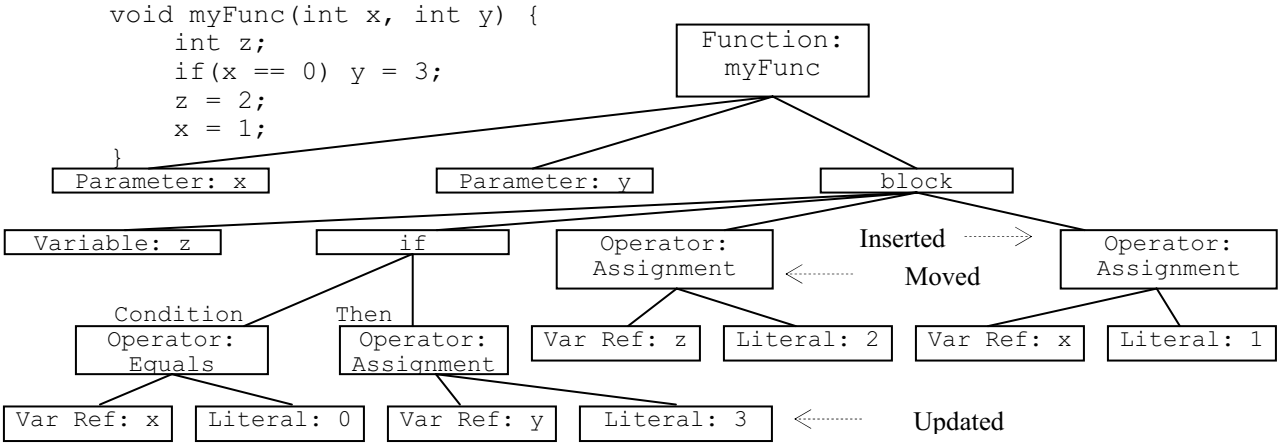

Figure 2: Modified version of function in Figure 1 and corresponding AST. Annotations indicate edit operations according to *Dex*.

The *DifferenceItemizer* analyzes the edit script to gather data about the differences between the two ASGs, creating an *itemization* or *profile* that is returned to the *DifferenceAnalyzer* together with *metadata*. The *DifferenceAnalyzer* maintains a *MetaDataSet* to store information from each pair of files that might be relevant to processing other pairs of files. For example, metadata is stored that describes function calls and function declarations within each pair of files. This is necessary because a file can include a function call that refers to a function defined in a different file. Once all pairs of files in a patch have been analyzed, the *DifferenceAnalyzer* passes the resulting profiles and final *MetaDataSet* to the *Aggregator*, which combines the profiles, creating a single profile containing key-value pairs, and appends the metadata. The *Aggregator* outputs a properties file describing the changes in the patch.

At present, *Dex* consists of 14,181 lines of Java code, of which 6,822 lines implement the differencing algorithm, 4,380 lines implement itemization, and 2,979 lines implement classes used in both of these functions. Currently Dex handles GXL files representing C source code, but it is possible to extend it to handle other languages.

5. Differencing algorithm

In this section, we describe *Dex*'s complex graph matching algorithm in detail, for the benefit of readers interested in understanding, adapting, or improving it. (Readers who are interested mainly in applications of *Dex* may wish to skip this section.) To determine the differences between two abstract semantic graphs, we created a heuristic algorithm that is specialized for ASGs with *Datrix* semantics. The algorithm is based on comparing two *ordered*, *rooted trees*[3], extracted from the ASGs. We call these *abstract semantic trees* (*ASTs*). They are similar in structure to the more common abstract syntax trees, but have additional semantic information embedded in them, e.g. type information. ASTs are obtained by retaining *tree edges* from the original ASG,

[3] In an ordered, rooted tree, the children of each node are ordered.

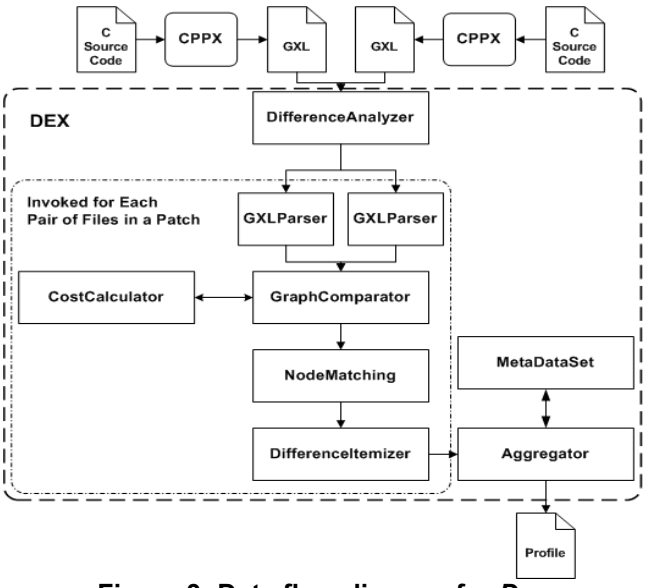

Figure 3: Data flow diagram for *Dex*

```
1 MATCHNODES
2       FINDMATCHESFROMROOT()
3       CREATENEWCOSTMATRICES()
4       WHILE (NUMBER OF REMAINING NODES > 0)
5               FINDEXACTMATCHES()
6       RECALCULATEIFNEEDED()
7       FINDINSERTSANDDELETES()
8               RECALCULATEIFNEEDED()
9               FINDUPDATES();
10              RECALCULATEIFNEEDED()
11              IF(NOCHANGESSINCEPREVIOUSPASS)
12                      BREAKNONZEROCOSTTIES()
13              RECALCULATEIFNEEDED()
14      POSTPROCESS()
15      RETURN NODEMATCHING

1 RECALCULATEIFNEEDED
2       IF(MATCHINGCHANGED)
3               FINDMATCHESFROMROOT()
4               CREATENEWCOSTMATRICES()
```

Figure 4: Top-level pseudocode.

by adding node attributes to represent the information given by non-tree edges, and by doing some additional cleanup of the graph. Our algorithm relies on iteratively matching parts of the ASTs and recalculating matching costs based on the whole ASGs, until all the nodes are matched. Note that this is different from the more commonly studied tree differencing problem [16][28][29] where costs are computed *a priori*. Our technique allows us to, for example, match variable references to a variable whose name changed, once the variable declaration is matched.

When our algorithm is applied to two ASTs T_1 and T_2, it produces an edit script describing how T_1 can be transformed into T_2 in a "natural" way. Because of the heuristic nature of the algorithm, this script is not guaranteed to have minimal length. It consists of a sequence of *match*, *insert*, and *delete* operations applied to particular nodes of the two trees. Each operation has an associated cost (see Section 5.2.1). *Match* operations, which map a node of T_1 to a node of T_2, are of four types. An *update* is a match between nodes that are not identical, e.g., nodes representing a literal whose value has changed. A *simple match* is a match between two nodes n_1 and n_2 such that the parents of n_1 and n_2 also match and the sibling ranks of n_1 and n_2 are the same. A *move* that *changes parent* is a match between nodes whose parents are not matched to each other. A *move* that *changes order* is a match between two nodes with matching parents but different sibling ranks. Other algorithms for computing edit distances between ordered trees [16][28] do not include a *move* operation. We felt this operation was necessary for analyzing code changes, because programmers sometimes move a statement or set of statements from one place to another. Since moves are not frequent, however, preference should be given to matches that do not result in a move. Adding a *move* operation complicates the differencing algorithm significantly, because it admits a much larger number of possible matches.

A node in an AST has a *type*, such as *literal*, *name reference* (variable name), **`for`**-*loop*, *logical operator* or *relational operator*. A node in T_1 will match a node in T_2 only if the two nodes have the same type, which simplifies cost calculations.

A top-level pseudocode description of the differencing algorithm is shown in Figure 4. On every iteration, the algorithm attempts to match everything that can be matched from the roots of ASTs T_1 and T_2 down toward their leaves, and then it attempts to match everything that can be matched from the leaves of the ASTs upward. Nodes in T_1 that do not match any node in T_2 are considered to be deleted; nodes in T_2 that do not match any node in T_1 are considered to be inserted. Between stages of the algorithm, the costs are recalculated, since matching nodes in the two trees with one another can change the cost of matching other nodes. In the remainder of this section, we present an overview of the principal aspects of the differencing algorithm. Further details can be found in [20].

5.1. Top-down matching algorithm

The purpose of the top-down matching is to quickly establish a rough correspondence between T_1 and T_2, to reduce the number of pairwise node comparisons that must be done. In the case studies described in Section 7, the number of nodes in typical ASTs ranged between

20,000 and 60,000,[4] whereas the number of nodes corresponding to *changed* code was only between 20 and 200 nodes. The top-down matching algorithm succeeded in matching 94% of all nodes on average, greatly reducing the overall running time of the differencing algorithm. For example, when comparing Figures 1 and 2, top-down matching will match all the nodes except for the assignment operator nodes and their subtrees.

The top-down matching algorithm does not consider the costs of edit operations. Starting at the roots of T_1 and T_2, this recursive algorithm finds potential matches between nodes one level at a time. On each recursive call, the children of two nodes $n_1 \in T_1$ and $n_2 \in T_2$ that are known to match are compared. A child of n_1 is considered to match a child of n_2 if there are paths of length four beginning with those children whose corresponding nodes have identical types and attributes. Children that match uniquely are added to a set of such nodes. Sometimes there are multiple children of n_1 that match a child of n_2 and/or multiple children of n_2 that match a child of n_1. In this case, a *tie-breaking heuristic* is employed, which considers the results of previous bottom-up matching. The algorithm checks if a child of n_1 has a descendant that matches a descendant of a child of n_2. If such children exist and are unique, they are considered uniquely matched. Otherwise the algorithm tries to break the tie by matching paths of length four. If this does not succeed, the tied nodes are left for later processing.

5.2. Bottom-up matching

Finding matches from the bottom up involves finding subtrees in T_1 identical to subtrees in T_2. For each node type τ, we use a dynamic programming algorithm to compute a *cost matrix* indexed by pairs of nodes of type τ, which indicates the costs of transforming T_1 into T_2 in different ways. The entry for $n_1 \in T_1$ and $n_2 \in T_2$ contains the minimum cost of an edit sequence that matches n_1 and n_2. The cost matrix also indicates the cost of inserting or deleting particular nodes. Entries of zero indicate exact matches of nodes and their entire subtrees.

5.2.1. Cost calculation. The rules for computing costs, which were refined by experimentation with a set of example AST pairs, are as follows:

- The total cost of matching two nodes of the same type is the sum of three terms: (1) a cost of 3 times the number of differences in their attributes and non-tree edges, (2) the added cost of optimally matching their children, and (3) a cost of 2 if the match is a move operation that changes parent.
- The cost of optimally matching two nodes with different numbers of children includes the cost of inserting or deleting children and their subtrees.
- In computing the optimal cost of matching the children of $n_1 \in T_1$ and $n_2 \in T_2$, the costs of matching n_1 with each child of n_2 and matching n_2 with each child of n_1 are considered.
- The cost of matching two nodes of different types is 20.
- The insert/delete cost is 2 per node.

Computing the cost of optimally matching the children of two nodes involves finding an optimal matching in a weighted bipartite graph. Note that our bottom-up matching algorithm allows the children of a node to be reordered, but at present it does not associate costs with such reordering.

Another consideration is the treatment of nodes that are strongly associated with their parent, as are many leaves and blocks. In an AST, leaves often correspond to literals and variable names, rather than to statements. Comparing a literal or variable to every other literal or variable, respectively, is unnecessary work and would produce spurious moves. Therefore, a leaf is compared only to the children of whatever its parent is compared to, unless the leaf's parent is a scope, in which case the leaf represents a statement. Similarly, a block associated with a function, conditional statement, or loop should be matched in conjunction with its parent. Such nodes are left out of the cost matrix.

5.2.2. Finding Exact Matches. When attempting to find exact matches between nodes of T_1 and T_2 (line 5 of Figure 4), the bottom-up matching algorithm looks for entries of zero in the cost matrix, which indicate not only that two nodes match exactly but also that their entire subtrees do. For a given node $n_1 \in T_1$, if the algorithm finds exactly one matching node $n_2 \in T_2$, it checks to be sure that n_2 matches exactly one node in T_1. If so, it adds these nodes and their subtrees to the matching. Otherwise, ties are broken by considering previous matches involving parents and siblings, and by considering depth from the root (see Section 5.2.5).

5.2.3. Finding Inserts and Deletes. A node n of type τ in T_1 or T_2 is marked as *deleted* or *inserted*, respectively, when there are no potential matches between n and nodes in the other tree. The bottom-up matching algorithm is conservative in deciding that there are no potential matches with n. This occurs only when either (1) there are no nodes of type τ in the other tree or (2) for each node m of type τ in the other tree for which the cost of matching n with m is minimal, both of the following conditions hold: (a) there is a unique node $k \neq n$ in n's

[4] Many of these nodes correspond to code from library header files (`#include` files).

tree for which the cost of matching k with m is minimal and (b) this cost is smaller than the cost of matching n with m (that is, n is not an optimal match for m). When comparing Figures 1 and 2, this portion of the algorithm matches the inserted subtree.

5.2.4. Finding Updates. Updates are pairs of nodes that match but have non-zero cost. For example, if the only change in a patch is to rename a variable in its declaration and all its uses, the variable declaration node will match but will be considered an update, because the name attribute is different, while the variable uses will be exact matches, as the referenced declaration nodes match. From a high level point of view, the algorithm looks for pairs of nodes of the same type which have not been matched but have matching parents. Alternatively, it looks for pairs of nodes with a cost which is a minimum in both its row and its column. It then updates these costs in the cost matrix to be zero, so that they will be matched as exact matches during the next iteration. In Figures 1 and 2 the updated assignment statement will be found during this stage. No tiebreaking occurs during this stage. Note that line 12 of Figure 4 will break ties for nodes of nonzero cost, but this will only be done if no other progress has been made during this iteration.

5.2.5. Tiebreaking. Tiebreaking is complicated by the fact that arbitrary sections of each graph might have already been marked as inserted or deleted. We have developed heuristics that try to minimize the number of moves. For example, consider the case where a program is modified by copying a piece of code and pasting it to multiple places. In this case, a node n_1 in T_1 matches to multiple nodes in T_2. Our heuristics break the tie based on context, by first giving preference to nodes whose parent has been matched to n_1's parent, after skipping inserted or deleted nodes. If there are multiple such nodes, as is the case when code is duplicated in the same block, ties are broken based on siblings. In the case that none of the parents match, multiple heuristics are tried, which consider the parents' node type, siblings, depth from root, or to recursively break the tie among the parents. As with all heuristics, there are always situations that it does not handle properly. In the case studies described in Section 7, the incorrect matches which occurred were due to tiebreaking errors.

5.3. Postprocessing

Since the differencing algorithm is heuristic in nature, it may match, insert, or delete nodes *prematurely*, in the sense that subsequent operations may alter costs in ways that invalidate the earlier ones. To address this, the algorithm ends with a *postprocessing* phase that uses additional heuristics to find and rectify such mistakes (line 14 of Figure 4). They are often indicated by numerous changes to a small part of an AST.

5.4. Time and space complexity

The worst-case complexity of the differencing algorithm is $O(n^4)$ and its space complexity $O(n^2)$, where n is the total number of nodes in the ASGs that are analyzed. The ASTs are created in linear time. Finding matches from the root makes four passes over all the nodes, as matches are verified to a depth of four. For each pass, the time required to compare each set of children is $O(nk^2)$, where k is the largest number of identical children per node. Cost calculation requires time and space on the order of the number of compared nodes. This number is equal to the total number of entries in all cost matrices, which is $O(n^2)$ in the worst case. Finding matches, insertions, deletions, or updates entails the following operations for each node r in one AST: (1) finding the minimum-cost entries in r's row of the cost matrix for r's node type and (2) checking whether those entries are the smallest in their columns. This requires $O(n^3)$ comparison operations in total. With each pass, the number of nodes that haven't been matched, inserted, or deleted must decrease by at least one, until there are none left. Therefore, the loop in Figure 4 is iterated at most n times and hence the total running time for the loop is at worst $O(n^4)$. Postprocessing makes an undetermined number of passes through all the nodes, but it will be at most mn where m is some constant, making postprocessing occur in $O(n^2)$ time. This algorithm will take at worst $O(n^4)$. The largest amount of space is taken by the cost matrices and the graphs, which will be at worst $O(n^2)$.

6. Itemizing differences

The *DifferenceItemizer* and *Aggregator* can be thought of as implementing an automated *survey* of the patches applied to a code base. In this survey, a set of questions about the individual changes comprising each patch is answered. Currently, the *DifferenceItemizer* and *Aggregator* record statistics about certain code changes that we judged to be useful for understanding the nature of *bug fixes*, although these statistics are also relevant to adaptive or perfective maintenance. Separate statistics are kept for changes involving different types of constructs. Changes involving conditional statements are analyzed in particular detail, due to our interest in missing condition defects (see Sections 2 and 7). The *DifferenceItemizer* analyzes changes in the context of an entire patch and, in several cases, in the context of the program constructs in which they occur. For example,

separate (sub)profiles are kept for changes made to branch conditions, function calls, and assignment statements. Each program construct is represented by a *subtree* of an AST. For our purposes, a *new* construct is one whose root has been marked as an *insert*. An *existing* construct is one that has been marked as a *match* to a node in the original graph, which includes *exact match*, an *update*, or a *move*. A construct is considered to be *altered* if it or any node in its subtree is marked as an *insert*, *delete*, *update*, or *move*.

The questions answered by the *DifferenceItemizer* range from simple to complex. Examples of the former include counts of how many existing function bodies were altered, how many new functions were inserted, how many function signatures changed, and how many function calls, assignment statements, conditional statements, loop statements and others statements were added, removed and modified. Examples of more complex questions include how many **`if`** conditions were altered so as to add a use of a variable that was already in scope and how many of these changes occurred for each kind of scope (function parameter, local variable, global variable). Notice that a textual difference does not provide enough information to answer this last question, which requires access to the semantic information contained in the ASG. *Dex* currently collects 398 statistics, and it is not difficult to modify it to collect different ones if needed. Further details about itemization of differences and a complete list of the statistics currently gathered can be found in [21].

7. Case studies

To evaluate the accuracy, performance, and usefulness of *Dex*, we used it to analyze bug fixes from two large open-source software development projects, the *Apache* HTTP server and the *GCC* compiler suite. For these case studies, we retrieved all the pertinent changes from the projects' mailing lists and CVS repositories, and then wrote scripts to automatically feed these to *Dex*. The *Apache* HTTP Server [1] is an open-source server developed for use with operating systems such as *UNIX* and *Windows NT*. For this study we selected seven months of changes made to version 2.0, spanning April 2002 through November 2002. Only those patches that included the (case insensitive) terms "bug", "fix", "fixed", or "fixes" were considered. We examined the logs for these patches by hand to remove those that were clearly not bug fixes. The resulting set contains 112 patches affecting 141 source files. *GCC* (*Gnu Compiler Collection*) [10] is an open-source suite of compilers for *UNIX*-compatible operating systems. For this study, we chose to examine the changes made to the *GCC C*-compiler between releases 3.2 and 3.2.1. According *GCC*'s change log, most of these changes were bug fixes. Those that were not were ignored, leaving 71 patches, which affected 95 source files.

7.1. Accuracy

Because *Dex* employs a heuristic differencing algorithm, its output is unlikely to be perfectly accurate. To evaluate the accuracy of *Dex*'s output, we selected at random 34 patches from the 71 for *GCC* and 39 patches from the 112 for *Apache*, and we checked *Dex*'s output for these patches manually.[5] We found that *Dex* produced output with some incorrect counts for 3 *Apache* patches and 6 *GCC* patches. There was exactly one incorrect count for each of the 3 *Apache* patches. For the 6 *GCC* patches, the number of incorrect counts ranged from 2 to 7, with an average of 3.5. Of the 398 counts gathered by *Dex*, 378 were always correct, 14 were incorrect in 1.4% of all patches, 3 were incorrect in 2.7% of the patches, 2 were incorrect in 4.1% of the patches, and one was incorrect in 5.5% of the patches. The average amount by which *Dex* was incorrect was 1.1.

7.2. Performance

In this study, *Dex* was run under the Windows 2000 Server operating system on a 1.8 GHz Pentium IV Xeon processor with 1 GB of RAM. Figure 5 shows the running time of the differencing algorithm, which dominates *Dex*'s overall running time, on the *Apache* and *GCC* files, as a function of the total number of nodes in the two ASGs that are compared. With the *Apache* files, the running time grows almost linearly until the total number of nodes reaches about 40,000. With the *GCC* files, the running time grows almost linearly until the total number of nodes reaches about 100,000. For both projects the running time of the differencing algorithm spikes for some of the largest ASGs; we are unsure of the cause of this phenomenon. For the *Apache* files, both the mean and median numbers of nodes per pair of files are about 30,000, and the corresponding running times are about 60 seconds. The *GCC* files give rise to much larger graphs: the mean number of nodes is about 75,000, and the median is about 64,000. Consequently, average running time of the differencing algorithm for the *GCC* files is about 5 minutes. Note that 67% of the *GCC* ASGs contain over 50,000 nodes, in comparison to only 13% of the *Apache* ASGs. We note that the performance *Dex*'s differencing algorithm can probably be improved substantially with tuning.

[5] This process took several person days.

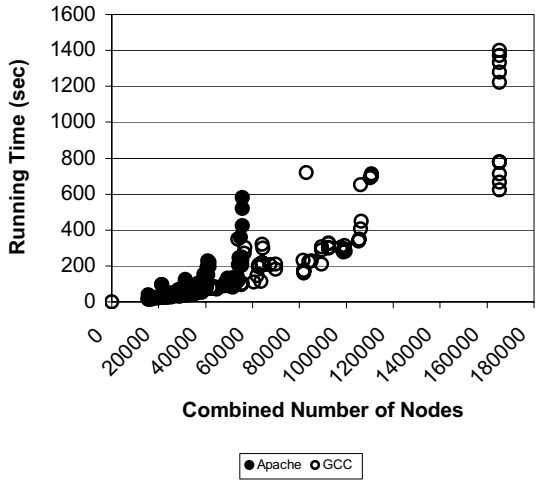

Figure 5: Running time vs. number of nodes

7.3. Analysis of results

Table 1 shows the most common type of changes for *Apache* and *GCC* and their frequencies. The six most common changes are the same for both projects, but in different order, which is somewhat surprising given the differences in problem domain and coding style. On both projects, the frequencies for the next most common types of changes drop sharply. The frequencies for changes that alter control flow by inserting a conditional statement (**`if`**, **`switch`**, or **`?`** operator) or by changing an existing **`if`** condition are of special interest to us, because such changes indicate attempted fixes of missing condition defects. *Dex* collects additional statistics about these changes, which permit a more detailed analysis. A large percentage of changes to **`if`** conditions added a new variable (62.32% for Apache, 47.50% for GCC). Over 25% of these changes added variables that were not even in scope beforehand (30.23% for *Apache* and 26.32% for *GCC*). For those **`if`** conditions that add only variables already in scope, Table 2 shows the frequency of the different types of variables inserted. There are large differences between the projects, but in both it was most common to add uses of local variables or of fields of variables already in scope. Similar results hold with respect to variables used in the branch conditions of inserted conditional statements (see [21] for details). To our knowledge, these are the first detailed statistics to be presented in the literature that characterize missing condition defects in major software products. Statistics characterizing other types of *Apache* and *GCC* bug fixes are addressed in [21].

The aforementioned statistics have important implications for how revisions of *Apache* and *GCC* should be validated. They indicate that missing condition defects are a major problem in these projects. Such defects present a difficult challenge for software testers. They are triggered only where certain relations involving program variables hold at particular locations in the code, yet it is often the case that the relations and the locations where they must hold are not specified in any project documents, presumably because they involve overlooked cases. *Dex* provides valuable clues about these relations and locations: most of the relevant variables are already in scope, though a significant number are not; many of the relations can be expressed by modifying expressions already present in the code; when a variable is missing from such an expression, it is usually a local variable or a field. In future work, we intend to modify *Dex* to further pin down the nature and location of typical missing condition defects, so that validation techniques can be designed specifically to reveal them.

8. Related work

Horwitz proposed a technique for identifying semantic and textual differences between program versions that is based on partitioning program components into sets of components with equivalent behaviors [14]. The partitioning algorithm represents programs using *program representation graphs*, which combine aspects of program dependence graphs and static single assignment forms but do not contain the kind of semantic information in ASGs (e.g., type information). The algorithm applies to programs in a limited language without procedures or functions. Binkley used a similar approach to define an algorithm for eliminating unnecessary regression tests and for reducing the size of the program that must be tested, although his algorithm applies to a language with procedures [4]. Jackson and Ladd proposed documenting the semantic difference between two versions of a procedure by comparing the dependence relations between input and outputs before and after the change [15]. Berzkins shows how the semantics of a program can be modeled by partial functions, and sets up Boolean and Browerian algebras to define what is meant by adding and removing functionality in different version of a program [3]. He then uses this model to study the problem of merging two sets of changes.

Krinke presents a technique for detecting duplicate code in a program based on detecting pairs subgraphs with identical length-*k* paths in a fine-grained program dependence graph [17],. A fine-grained PDG has nodes similar to those on an AST, and edges corresponding to those in the AST plus control and data dependence edges. Analyzing this graph allows the algorithm to find similar subgraphs based on data dependences, even if the control

Table 1: Frequency of the six most common types of changes in both GCC and Apache, as percentage of patches

Type of change	Apache	GCC
Altered existing function bodies	94.64%	90.14%
Inserted conditional statements into existing functions	37.50%	43.66%
Inserted function calls	37.50%	56.34%
Altered existing function calls	33.04%	26.76%
Altered existing `if` conditions	31.25%	32.39%
Altered existing assignment statements	25.89%	36.62%

dependences have changed. The end result is a many-to-many matching of subgraphs of size at most *k*, and not all parts of the graph are matched. This is useful for finding duplicated code, and Krinke shows that this produces few false positives and has high accuracy.

Yang described an algorithm for finding syntactic differences for use with version control software between two programs that works by analyzing their *parse trees* [27]. Unlike our algorithm, Yang's parse trees do not contain the semantic information present in ASTs and his algorithm places restrictions on admissible matchings that our algorithm does not: (1) two nodes may only match if their parents match and (2) the order of sibling nodes must be preserved. Wang *et al* describe a binary matching tool called *BMAT* [25]. This tool works on program binaries, by matching basic blocks from the original binary to basic blocks in the new one. Matching is done based on similarity of basic blocks and limited control flow information. Software differencing can also be aided with support from the editing tool. Berlage and Genau propose representing differences by the sequence of commands executed by the user to make the change, which are automatically recorded by the application [2]. They also study how to merge two command sequences.

A well-studied application of software differencing is *software merging*, where two sets of changes to a program have to be merged into a single final version. One of the first algorithms to provide guarantees on the behavior of the merged program is given by Binkley *et al* in [5]. Mens surveys current results on software merging, including an overview of several differencing algorithms currently used for software merging [19], most of which are presented in this section. Westfechtel presents an approach for merging of revisions where the editor is aware of the AST of the document and automatically assigns tags to new and modified nodes, making it trivial to find the differences between versions [26].

Zhang and Shasha review older work on tree differencing and present a fast algorithm for solving the edit-distance problem on ordered trees, with updates, insertions, and deletions [28]. These classical algorithms require all costs to be known beforehand, and cannot

Table 2: Variables added to `if` conditions. As percentage of total `if` conditions modified.

Type of variable added	Apache	GCC
Local variable	66.67%	64.29%
Field	60.00%	57.14%
Function Parameter	30.00%	14.29%
Global variable	3.33%	21.43%

handle changing costs for parts of the trees based on whether other parts match. Chawathe and Garcia-Molina presented an algorithm to find changes in structured data modeled by unordered trees, which also includes *move, copy* and *glue* operations [6]. Their algorithm is a heuristic, iterative update mechanism like ours, but only accommodates tree edges. Note that the edit-distance problem between unordered trees is, in general, NP-complete [29].

Changes from a source control repository can be analyzed without using software differencing. Graves and Graves and Mockus investigated the possibility of determining effort spent on different changes by analyzing size and type of each change, together with reported total monthly effort for each developer from an accounting database [12]. Using this technique they looked for modules that were becoming harder to maintain, and also compared the amount of effort necessary for bug fixes versus new features.

9. Conclusion

Dex provides an automated means to collect detailed information about the nature of code changes in large software projects, including syntactic changes and certain semantic ones. The case studies reported in Section 7 demonstrate that *Dex* can provide information that is valuable for evaluating the applicability of certain software testing techniques to a project. In these studies, *Dex* showed good accuracy and acceptable performance, although its matching algorithm would benefit from further tuning. We believe that with appropriate modifications, *Dex* can provide information that is useful for evaluating a variety of software engineering techniques related to software maintenance. An interesting topic for future work is how *Dex* can be adapted to detect higher-level code changes, such as ones that crosscut multiple modules or that involve dependences between non-contiguous program elements. Such extensions to *Dex* are likely to require the integration of other semantic differencing techniques such as ones based on program dependence analysis (see Section 8).

10. Acknowledgments

This work was supported by National Science Foundation award CCR-0098325 to Case Western Reserve University.

11. References

[1] The Apache Software Foundation. Apache HTTP Server. http://httpd.apache.org (accessed January 2003).

[2] T. Berlage and A. Genau. A Framework for Shared Applications with Replicated Architectures. Proceedings of the 6th annual ACM symposium on user interface software and technology. December 1993.

[3] V. Berzins. "Software Merge: Semantics of Combining Changes to Programs." ACM Trans. Programming Languages and Systems, 16(6):1875-1903, 1994

[4] D. Binkley. Using semantic differencing to reduce the cost of regression testing. 1992 International Conference on Software Maintenance (Orlando, FL, November 1992), 41-52.

[5] D. Binkley, S. Horwitz, T. Reps. "Program Integration for Languages with Procedure Calls" ACM Transactions on Software Engineering and Methodology, 4(1): 3-35, 1995

[6] S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. 1997 ACM SIGMOD International Conference on Management of Data (Tucson, AZ, 1997), 26-37.

[7] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong. Orthogonal defect classification: a concept for in-process measurement. IEEE Transactions on Software Engineering 18, 11 (November 1992), 943-956.

[8] Cvshome.org. Concurrent Versions System. http://www.cvshome.org.

[9] W. Dickinson, D. Leon, and A. Podgurski. Pursuing failure: the distribution of program failures in a profile space. 10th European Software Engineering Conference and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (Vienna, September 2001), 246-255.

[10] Free Software Foundation. GNU Compiler Collection. http://www.gnu.org (accessed January 2003).

[11] Gnu.org. diffutils. http://www.gnu.org/software/diffutils/diffutils.html.

[12] T. Graves and A. Mockus. Inferring Change Effort from Configuration Management Databases. Proceedings of the International Symposium on Software Metrics. 1998.

[13] R. Holt, A. Shur, S. E. Sim, and A. Winter. GXL: graph exchange language. http://www.gupro.de/GXL/.

[14] S. Horwitz. Indentifying semantic and textual differences between two versions of a program. ACM SIGPLAN '90 Conference on Programming Language Design and Implementation (White Plains, NY, June 1990), 234-245.

[15] D. Jackson, D.A. Ladd. "Semantic Diff: A tool for summarizing the effects of modifications." Proc. Int'l Conf. Software Maintenance, 1994.

[16] P. N. Klein. Computing edit distance between unordered rooted trees. 6th European Symposium on Algorithms (Venice, August 1998), Lecture Notes in Computer Science 1461, Springer Verlag, 91-102.

[17] J. Krinke. Identifying similar code with program dependence graphs. 8th Working Conference on Reverse Engineering (Stuttgart, Germany, October 2001), 201-209.

[18] D. Leon, A. Podgurski., and L. J. White. Multivariate visualization in observation-based testing. 22nd International Conference on Software Engineering (Limerick, Ireland, June 2000), 116-125.

[19] T. Mens. "A Survey on Software Merging.", IEEE Trans. Software Engineering, 28(5), May 2002.

[20] S. Raghavan. Finding differences in program code through abstract semantic graph comparison. M.S. project report, available at http://softlab4.cwru.edu/dex/index.html.

[21] R. Rohana. Analysis of common programmer error via semantic differencing. M.S. project report, available at http://softlab4.cwru.edu/dex/index.html.

[22] Software Architecture Group (SWAG) (2002). CPPX. University of Waterloo, http://swag.uwaterloo.ca/swagkit (accessed January, 2003).

[23] Software Architecture Group (SWAG) (2001). Original CPPX and Documentation. University of Waterloo, http://swag.uwaterloo.ca/~cppx (accessed August, 2002).

[24] F. I. Vokolos and P. G. Frankl. Pythia: a regression test selection tool based on textual differencing. 3rd International Conference on Reliability, Quality, and Safety of Software Intensive Systems (May 1997).

[25] Z. Wang, K. Pierce, and S. McFarling. BMAT: a binary matching tool for stale profile propagation. Journal of Instruction-Level Parallelism 2 (May 2000).

[26] B. Westfechtel. "Structure-Oriented Merging of Revisions of Software Documents." Proc. Third Int'l conf. Software Configuration Management, 1991

[27] W. Yang. Identifying syntactic differences between two programs. Software Practice and Experience 21, 7 (July 1991), 739-755.

[28] K. Zhang and D. Shasha. Approximate tree pattern matching. In: Pattern Matching Algorithms, edited by A. Apostolico and Z. Galil, Oxford University Press, 1997.

[29] K. Zhang, R. Statman, and D. Shasha. On the editing distance between unordered, labeled trees. Information Processing Letters 42 (1992), 133-139.