# Program Element Matching for Multi-Version Program Analyses

Miryung Kim, David Notkin
Computer Science & Engineering
University of Washington
Seattle, WA
{miryung,notkin}@cs.washington.edu

## ABSTRACT

Multi-version program analyses require that elements of one version of a program be mapped to the elements of other versions of that program. Matching program elements between two versions of a program is a fundamental building block for multi-version program analyses and other software evolution research such as profile propagation, regression testing, and software version merging.

In this paper, we survey matching techniques that can be used for multi-version program analyses and evaluate them based on hypothetical change scenarios. This paper also lists challenges of the matching problem, identifies open problems, and proposes future directions.

**Categories and Subject Descriptor:** D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement —*restructuring, reverse engineering, and reengineering*

**General Terms:** Documentation, Algorithms

**Keywords:** matching, software evolution, multi-version analysis

## 1. INTRODUCTION

In the last several years, researchers in software engineering have begun to analyze programs together with their change history. In contrast to traditional program analyses that examine a single version, multi-version program analyses use multiple versions of a program as input and mine change patterns.

There are roughly two different types of multi-version analyses: (1) coarse-grained analyses and (2) fine-grained analyses. Coarse-grained analyses compute changes between two consecutive versions of a program, aggregate the change information across multiple versions or across multiple files, and infer coarse-grained patterns [37, 15, 20, 17]. For example, Nagappan and Ball's analysis [37] finds line-level changes between two consecutive versions, counts the total number of changes per binary module, and infers the characteristics of frequently changed modules. On the other hand, fine-grained analyses track how individual code fragments changed during program evolution to infer fine-grained change patterns [29, 31, 52, 38, 43, 48, 51, 11]. For example, a clone genealogy extractor tracks individual code snippets over multiple versions to infer clone evolution patterns [29]. As another example, a signature change pattern analysis [31, 30] traces how the name and the signature of functions change. Matching elements between two versions of a program is a fundamental building block for fine-grained multi-version analyses as well as other software evolution research such as software version merging, regression testing, profile propagation, etc [36, 42, 21, 46].

We first define the problem of matching code elements between two versions:

*Suppose that a program $P'$ is created by modifying $P$. Determine the difference $\Delta$ between $P$ and $P'$. For a code fragment $c' \in P'$, determine whether $c' \in \Delta$. If not, find $c'$'s corresponding origin $c$ in $P$.*

The problem definition states that we must compute the difference between two programs. Computing semantic differences requires solving the problem of semantic program equivalence, which is an undecidable problem. Thus, once the problem is approximated by matching a code element by its syntactic and textual similarity, solutions depend on the choices of (1) an underlying program representation, (2) matching granularity, (3) matching multiplicity, and (4) matching heuristics. In this paper, we explain how the choices impact applicability of each matching method and how the choices affect effectiveness and accuracy of matching by creating an evaluation framework for existing matching techniques.

The rest of this paper is organized as follows. The next section discusses several multi-version analyses, which demonstrate the needs of program element matching. Then, we discuss challenges of program element matching in Section 3. Section 4 presents a survey of state-of-the-art matching techniques from various research areas such as multi-version program analyses, profile propagation, software version merging, and regression testing. Section 5 compares the surveyed techniques and Section 6 evaluates each technique using hypothetical program change scenarios. Section 7 presents open problems and future directions.

## 2. MOTIVATING PROBLEMS

In this section, we describe several multi-version analysis problems that demonstrate importance of program element matching.

The first problem is maintaining two similar programs


*MSR'06,* May 22–23, 2006, Shanghai, China.

that originated from the same source but evolved differently in parallel.[1] In many organizations, it is a common practice to clone a product or a module and maintain the clones in parallel [13]. Maintenance difficulties arise when programmers discover a critical bug in cloned parts. If programmers do not know whether the discovered bug is relevant to other cloned counterparts, they must inspect source code of the counterparts. We believe that programmers can better locate relevant counterparts by understanding how clones change over time. Monitoring clones requires tracking each clone by its physical location such as a file name, a procedure name, or calibrated line numbers [29].

Our second motivating problem is understanding the evolution of information hiding interfaces. The information hiding principle [41] states that programmers should anticipate what kinds of decisions are likely to change in the future and hide them using an interface. In general, it is difficult for programmers to predict which design decisions are likely to change; thus, unanticipated design decisions result in degradation of original software design. We believe that understanding interface evolution can shed light on (1) which decisions were originally hidden but later exposed and (2) how unanticipated decisions impact original interface design.

In addition to the problems above, type change analysis [38], signature change analysis [31], and instability analysis [11] also require matching program elements in order to track code elements over time.

# 3. MATCHING CHALLENGES

This section lists challenges of program element matching.

## 3.1 Absence of Benchmarks

It is difficult to evaluate matching techniques because there is no reference data set or archive of editing logs. Previous studies [30] also indicate that programmers often disagree about the origin of code snippets; low inter-rater agreement suggests that there may be no ground truth in program element matching.

## 3.2 Various Granularity Support

With respect to our motivating problems in Section 2, we cannot assume that programmers intend to track program elements at a fixed granularity. There are two reasons why matching techniques must support various granularity. First, a programmer may want to track design decisions that cannot be mapped to program units precisely. Second, a programmer may want track program elements at a different granularity depending on the nature of tasks. For example, if matching techniques are to be used for profile propagation or precise regression test selection, mappings should be found at a fine granularity such as at the level of control flow graph edges [42] or at the level of code blocks [46, 44]. On the other hand, if a programmer wants to use matching results for program understanding tasks, it would be more appropriate to find associations at a higher level such as a file.

## 3.3 Types of Code Changes

Certain types of code changes make the matching problem non-trivial. For example, tracking code by its enclosing procedure name would fail if programmers merged, split, or renamed procedures. When a programmer copies code, matching techniques cannot assume one-to-one mappings between old elements and new elements because an old code element can have more than one matching descendants in a new version.

[1] In an open source project community, this practice is often called as forking.

# 4. MATCHING TECHNIQUES

This section describes matching techniques used for software version merging, program differencing, profile propagation, regression testing, and multi-version program analyses. For easy comparison, we group techniques by program representation. We describe clone detectors and tools that infer refactoring events in Section 4.7 and 4.8 because these tools can be leveraged for finding correspondences between two versions.

## 4.1 Entity Name Matching

The simplest matching method treats program elements as immutable entities with a fixed name and matches the elements by name. For example, Zimmermann et al. modeled a function as a tuple, *(file name, FUNCTION, function name)*, and a field as a tuple, *(function name, FIELD, field name)* [51]. Similarly, Ying et al. [48] modeled a file with its full path name. In fact, matching by name would be sufficient for many multi-version analyses that intend to identify coarse-grained patterns such as the characteristics of fault prone modules [15, 20, 37].

## 4.2 String Matching

When a program is represented as a string, the best match between two strings is computed by finding the longest common subsequence (LCS) [7]. The LCS problem is built on the assumption that (1) available operations are addition and deletion, and (2) matched pairs cannot cross one another. Thus, the longest common subsequence does not necessarily include all possible matches when available edit operations include copy, paste, and move. Tichy [45] (*bdiff*) extended the LCS problem by relaxing the two assumptions above: permitting crossing block moves and not requiring one-to-one correspondence.

The line-level LCS implementation, *diff* [25], has served as basis for many multi-version analyses, because (1) *diff* is fast and reliable, and (2) popular version control systems such as CVS [2] or Subversion [1] already store changes as line-level differences. For example, a clone genealogy extractor tracks code snippets by their file name and line number [29]. As another example, fix-inducing code snippets [43] are inferred by tracking backward a tuple of *(file name:: function name:: line number)* from the moment that a bug is fixed.

## 4.3 Syntax Tree Matching

For software version merging, Yang [47] developed an AST differencing algorithm. Given a pair of two functions $(f_T, f_R)$, the algorithm creates two abstract syntax trees $T$ and $R$ and attempts to match the two tree roots. If the two roots match, the algorithm aligns $T$'s subtrees $t_1, t_2, ..., t_i$ and $R$'s subtrees $r_1, r_2, ...r_j$ using the LCS algorithm and matches subtrees recursively. This type of tree matching respects the parent-child relationship as well as the order between sibling nodes, but is very sensitive to changes in nested block and control structures because tree roots must be matched for every level.

Hunt and Tichy do not directly compare ASTs but use syntactic information to guide string level differencing. Their 3-way merging tool [24] parses a program into a language neutral form, compares token strings using the LCS algorithm, and finds syntactic changes using structural information from the parse.

For dynamic software updating, Neamtiu et al. [38] built an algorithm that tracks simple changes to variables, types, and functions based on a AST representation. Neamtiu's algorithm assumes that function names are relatively stable over time and matches the ASTs of functions with the same name; the algorithm traverses two ASTs in parallel and incrementally adds one-to-one mappings as long as the ASTs have the same shape. In contrast to Yang's algorithm, Neamtiu's algorithm cannot compare structurally different ASTs.

## 4.4 Control Flow Graph Matching

Laski and Szermer [33] first developed an algorithm that computes one-to-one correspondences between CFG nodes in two programs $P1$ and $P2$. This algorithm first reduces a CFG to a series of single-entry, single-exit subgraphs called hammocks and matches a sequence of hammock nodes using a depth first search (DFS). Once a pair of corresponding hammock nodes is found, the hammock nodes are recursively expanded in order to find correspondences within the matched hammocks.

*Jdiff* [5] extends Laski and Szermer's (LS) algorithm to compare Java programs based on an enhanced control flow graph (ECFG). *Jdiff* is similar to the LS algorithm in the sense that hammocks are recursively expanded and compared, but is different in three ways: First, while the LS algorithm compares hammock nodes by the name of a start node in the hammock, *Jdiff* checks whether the ratio of unchanged-matched pairs in the hammock is greater than a chosen threshold in order to allow for flexible matches. Second, while the LS algorithm uses DFS to match hammock nodes, *Jdiff* only uses DFS up to a certain look-ahead depth to improve its performance. Third, while the LS algorithm requires hammock node matches at the same nested level, *Jdiff* can match hammock nodes at a different nested level; thus, *Jdiff* is more robust to addition of while loops or if-statements at the beginning of a code segment. *Jdiff* has been used for regression test selection [40] and dynamic impact analysis [6].

## 4.5 Program Dependence Graph Matching

There are several program differencing algorithms based on a program dependence graph [23, 12, 26]. These algorithms are not applicable to popular modern program languages because they can run only on a limited subset of C languages without global variables, pointers, arrays, or procedures.

## 4.6 Binary Code Matching

*BMAT* [46] is a fast and effective tool that matches two versions of a binary program without knowledge of source code changes. *BMAT* was used for profile propagation and regression test prioritization [44]. BMAT's algorithm matches blocks in three steps. The first step of BMAT's matching algorithm is to find one-to-one mappings between the procedures in two versions based on their names, type information, and code contents. To match procedures with different names, block trial matching is done on procedure pairs with a small number of character differences in their hierarchical names. In this step, the thresholds for procedure name difference and block matching percentage are both set heuristically. In the second step, BMAT first matches data blocks within each pair of matched procedures using a hash function and matches remaining unmatched data blocks if the unmatched blocks are sandwiched by already matched pairs. In the third step, BMAT matches code blocks in multiple hashing passes. During hash-based matching, if hash values collide, two heuristics are used to break ties: (1) crossing matches are forbidden at certain hashing passes, and (2) a pair of blocks is preferred to other tied pairs if either its predecessors or successors are also matched. For remaining unmatched blocks, BMAT matches blocks based on control flow equivalence, allowing one-to-many mappings between old code blocks and new code blocks.

## 4.7 Clone Detection

A clone detector is simply an implementation of an arbitrary equivalence function. The equivalence function defined by each clone detector depends a program representation and a comparison algorithm. Most clone detectors [8, 28, 9, 32, 27] are heavily dependent on (1) hash functions to improve performance, (2) parameterization to allow flexible matches, and (3) thresholds to remove spurious matches. A clone detector can be considered as a many-to-many matcher based solely on content similarity heuristics.

## 4.8 Origin Analysis Tools

Origin analysis infers refactoring events such as splitting, merging, renaming and moving by comparing two versions of a program [14, 52, 31, 4, 18, 35, 19]. Origin analysis tackles the program element matching problem directly but produces matching results only at a predefined granularity such as a procedure, class or file.

Demeyer et al. [14] first proposed the idea of inferring refactoring events by comparing the two programs. Demeyer et al. used a set of ten characteristics metrics for each class, such as LOC and the number of method calls within a method (i.e., fan-out) and inferred where refactoring events occurred based on the metric values and a class inheritance hierarchy.

Zou and Godfrey's origin analysis [52] matches procedures using multiple criteria (names, signatures, metric values, and a set of callers and callees) and infers merging, splitting, and renaming events. Both Demeyer et al. and Zou and Godfrey's analyses are semi-automatic in the sense that a programmer must manually tune matching criteria and select a match among candidate matches.

Kim et al. [30] automated Zou and Godfrey's procedure renaming analysis. In addition to matching criteria used by Zou and Godfrey, Kim et al. used clone detectors such as CCFinder [28] and Moss [3] to calculate content similarity between procedures. An overall similarity is computed as a weighted sum of each similarity metric, and a match is selected if the overall similarity is greater than a certain threshold. To create an evaluation data set, ten human judges identified renaming events in the Subversion and the Apache projects, and if seven out of ten judges agreed the origin of a renamed procedure, a match was added to a reference data set. Using the reference data set, Kim et al. optimized each factor's weight and tuned the threshold

**Table 1: Comparison of Matching Techniques**

| Program Representation | Citation | Granularity | Assumed Correspondence | Multiplicity | Heuristics N | Heuristics P | Heuristics S | Application |
|---|---|---|---|---|---|---|---|---|
| A set of entities | [20, 15, 37] | Module | | 1:1 | √ | | | Fault proneness |
| | Bevan et al. [11] | File | | 1:1 | √ | | | Instability |
| | Ying et al. [48] | File | | 1:1 | √ | | | Co-change |
| | Zimmermann et al. [51] | File Procedure Field | | 1:1 | √ | | | |
| String | *diff* [25] | Line | File | 1:1 | | | √ | Merging Clone genealogy [29] Fix inducing code [43] |
| | *bdiff* [45] | Line | File | 1:n | | | √ | Merging |
| AST | *cdiff* [47] | AST Node | Procedure | 1:1 | √ | | | |
| | Neamtiu et al.[38] | Type, Var | | 1:1 | √ | √ | | Type change |
| | Hunt, Tichy[24, 35] | Token | File | 1:1 | | √ | √ | Merging |
| CFG | *Jdiff* [5] | CFG node | | 1:1 | √ | | √ | Regression testing Impact analysis |
| Binary | BMAT [46] | Code block | | 1:1 (procedure) n:1 (block) | √ | √ | √ | Profile propagation Regression testing |
| Hybrid | Zou, Godfrey [52] | Procedure | | 1:1 or 1:n or n:1 | √ | | √ | Origin analysis |
| | Kim et al. [30] | Procedure | | 1:1 | √ | | √ | Signature change [31] Renaming analysis |

N: Name-based heuristics, P: Position-based heuristics, S: Similarity-based heuristics

value. The accuracy of Kim's tool was better than the average accuracy of human judges, indicating that human judges significantly disagreed about the origin of procedures.

## 5. COMPARISON

Table 1 shows comparison of the state-of-the-art matching techniques in Section 4. As shown in the fourth column of Table 1, many matching techniques assume correspondence at a certain granularity no matter whether this assumption is explicitly stated or not. For example, using *diff* to match code snippets assumes that input files already are matched. As another example, using *cdiff* to match AST nodes assumes that enclosing functions are matched by the same name.

All matching techniques heavily rely on heuristics to reduce a matching scope and to improve precision and recall. The heuristics are categorized into three categories: [2]

1. Name-based heuristics match entities with similar names. For example, BMAT and Jdiff match procedures in multiple phases by the same globally qualified name (e.g., System.out.println), by the same hierarchical name, by the same signature, and by the same name.
2. Position-based heuristics match entities with similar positions. If entities are placed in the same syntactic position or surrounded by already matched pairs (i.e., a sandwich heuristic), they become a matched pair. For example, BMAT uses a sandwich heuristic aggressively to remove unmatched pairs. As another example, Neamtiu's algorithm traverses two ASTs in parallel and matches variables placed in the same syntactic position regardless of their labels.
3. Similarity-based heuristics match entities that are nearly identical; they often rely on parameterization and a hash function to find near identical entities. All clone detectors can be viewed as a similarity-based matcher.

[2]The three categories are not comprehensive or mutually exclusive.

The three different types of heuristics complement one another. For example, when hash values collide or parameterization results in spurious matches, position-based heuristics will select a matched pair that preserves linear ordering or structural ordering by checking neighboring matches. Table 1 (column 6 to 8) summarizes which kinds of heuristics that each matching technique uses.

## 6. EVALUATION

Matching techniques are often inadequately evaluated—Only Kim et al. conducted a comparative study using human subjects [30]. This lack of evaluation is exacerbated by the fact that there are no agreed evaluation criteria or representative benchmarks. Finding such universal criteria would be difficult since each technique is built for a different goal. For example, matching techniques for regression testing or profile propagation [5, 46, 49] can be evaluated by the accuracy of static branch prediction and code coverage; but even this evaluation method is not applicable to programs without test suites. To evaluate matching techniques uniformly, we take a scenario-based evaluation approach; we design a small set of hypothetical program change scenarios, on which we describe how well various matching techniques will perform. [3]

*Scenario 1: (1) a programmer inserts if-else statements in the beginning of the method $m_A$, and (2) the programmer reorders several statements in the method $m_B$ without changing semantics as shown in Table 2.*

The ideal matching technique should produce (s1-s1'), (s2-s2'), (s3-s4'), (s4-s3'), and (s5-s5') and identify that s0' is added. The third column of Table 3 summarizes how well each technique will work in this scenario. *Diff* can match lines of $m_A$ but cannot match reordered lines in $m_B$ because the LCS algorithm does not allow crossing block moves. On the other hand, *bdiff* can match reordered lines in $m_B$ because crossing block moves are allowed. Neamtiu's algo-

[3]PDG-based matching techniques are excluded due to lack of modern programming language support.

**Table 2: Scenario 1 Code Change**

| before | after |
|---|---|
| mA (){<br>  if (pred_a) { \\s1<br>  foo() \\s2<br>  }<br>}<br>mB (b){<br>  a:= 1 \\s3<br>  b:= b+1 \\s4<br>  fun(a,b) \\s5<br>} | mA (){<br>  if (pred_a0) { \\s0'<br>  if (pred_a) { \\s1'<br>  foo() \\s2'<br>  }<br>  }<br>}<br>mB (b){<br>  b:= b+1 \\s3'<br>  a:= 1 \\s4'<br>  fun(a,b) \\s5'<br>} |

rithm will perform poorly in both $m_A$ and $m_B$ because it does not perform a deep structural match. *Cdiff* cannot match unchanged parts in $m_A$ correctly because *cdiff* stops early if roots do not match for each level. *Jdiff* will be able to skip the changed control structure, map unchanged parts in $m_A$, and match reordered statements in $m_B$ if the look-ahead threshold is greater than the depth of nested controls. *BMAT* cannot track code blocks in $m_B$ because BMAT's hashing algorithms are instruction order sensitive. In conclusion, *Jdiff* will work the best for changes within procedures at a statement or predicate level.

*Scenario 2: A file PElmtMatch changed its name to PMatching. A procedure matchBlck are split into two procedures matchDBlck and matchCBlck. A procedure matchAST changed its name to matchAbstractSyntaxTree.*

The ideal matching technique should produce (PElmtMatch, PMatching), (matchBlck, matchDBlck), (matchBlck, matchCBlck), and (matchAST, matchAbstractSyntaxTree). The fourth column of Table 3 summarizes how each technique will work in this scenario. Most name-based matching techniques will do poorly due to renaming events. *Diff* and *bdiff* will be able to track each line only if file names did not change. Both *cdiff* and Neamtiu's algorithm will perform poorly if procedure names changed. Both *BMAT* and origin analysis tools will perform well because they rely on multiple passes of hash functions and multiple phases of name matching.

The remaining columns of Table 3 describe how well each matching technique will work in case of restructuring tasks at a procedure level or at a file level.

Based on Table 1 and 3, we conclude the following:

- Matching techniques based on AST or CFG produce matches at fine-grained levels but are only applicable to a complete and parsable program. Researchers must consider the trade-off between matching granularity, matching requirements, and matching cost.
- Many techniques employ the LCS algorithm even when matching AST or CFG, thus inheriting the assumptions of LCS: one-to-one correspondences between matched entities and linear ordering among matched pairs. This sort of implicit assumptions must be carefully examined before implementing a matcher.
- Most techniques support only one-to-one mappings at a fixed granularity. Therefore, they will perform poorly when merging or splitting occurs.
- The more heuristics are used, the more matches can be found by complementing one another. For example, name-based matching is easy to implement and can reduce matching scope quickly, but it is not robust to renaming events. In this case, similarity-based matching can produce matches between renamed entities and position-based matching can leverage already matched pairs to infer more matches.

# 7. FUTURE DIRECTIONS

This section lists remaining open problems and future directions.

**Hybrid Matcher.** Although no single technique performs perfectly in all change scenarios but the combination of all techniques does. Thus combining multiple techniques may improve the accuracy of matches by complementing one another. The simplest way is to run all matching techniques separately and find consensus among the results. Another way of building a hybrid matcher is to leverage a feedback loop between matching tools and tools that infer refactoring events. Determining which refactoring occurred and determining correspondences is a chicken and egg problem; inferring refactoring events requires knowledge of correspondences, and finding good correspondences is achieved by knowing which refactoring occurred. This feedback cycle provides an opportunity to find more matches. The results of inferred transformations are fed to matching tools, and the matching results are fed back to a refactoring reconstruction tool iteratively until optimal correspondences are found.

We must note that combining results from multiple matchers will require tremendous efforts because (1) not every matching tool is available for public use or applicable to popular programming languages and (2) different matchers use different program representations.

**Capturing Editing Operations.** Having a complete history of logical editing operations would nullify the matching problem. However, most software repositories employ state-based merging not operation-based merging [34], thus making it impossible to restore logical editing operations completely. Even when an edit log is available, if editing operations are captured at a key stroke level, it is not trivial to reconstruct logical editing operations (such as procedure renaming, splitting, and merging) and produce matches between program elements. Recently, capturing and replaying refactoring operations is shown to be possible in an Eclipse IDE [22], so we can leverage this type of refactoring history to initiate the feedback loop discussed above.

**Interval Manipulation vs. Matching Tool Selection.** In this paper, we simplified a multi-version program matching problem as a two version matching problem. To use a matching technique in the context of multi-version analyses, the interval between each pair of versions must be determined. In the past, the granularity of available historical data limited a sampling interval for multi-version analyses. Recently, several infrastructures [10, 51, 50] were built to facilitate multi-version analyses by restoring commit transactions from a source code repository and automatically extracting multiple versions separated by an arbitrary time interval. These infrastructures enable multi-version analyses to manipulate a sampling interval. Therefore, the remaining problem is to determine an optimal sampling interval for each matching technique (or select an appropriate

**Table 3: Evaluation of the Surveyed Matching Techniques**

| Program Representation | Citation | Scenario | | Transformations | | | | Strength and Weakness |
|---|---|---|---|---|---|---|---|---|
| | | | | Split/Merge | | Rename | | |
| | | 1 | 2 | Proc | File | Proc | File | |
| String | *diff* [25] | ⊠ | □ | □ | □ | ⊠ | □ | − sensitive to file name changes |
| | *bdiff* [45] | ■ | □ | ⊠ | □ | ⊠ | □ | + can trace copied blocks |
| AST | *cdiff* [47] | □ | □ | □ | □ | □ | □ | − sensitive to nested level change<br>− require procedure level mappings |
| | Neamtiu et al. [38] | □ | □ | □ | □ | □ | □ | − partial AST matching |
| | Hunt, Tichy [24, 35] | ⊠ | □ | □ | □ | ■ | □ | − require file level mappings<br>+ can identify procedure renaming |
| CFG | *JDiff* [5] | ■ | ⊠ | □ | □ | ⊠ | ⊠ | + robust to signature changes<br>− sensitive to control structure changes |
| Binary | BMAT [46] | □ | ■ | □ | □ | ■ | ■ | + robust to procedure renaming<br>− assume 1:1 procedure correspondence<br>− only applicable to binary programs |
| Hybrid | Zou, Godfrey [52] | □ | ■ | ■ | ■ | ■ | ■ | − semi-automatic, manual analysis |
| | Kim et al. [30] | □ | ■ | □ | □ | ■ | ■ | − assume 1:1 procedure correspondence |

■ good ⊠ mediocre □ poor

matching tool depending on the logical gap between two versions of a program). Another interesting open question is, "can we design a matching technique that works as well as aggregating results from a set of program snapshots that separated by only small changes?"

**Matching Result Aggregation.** As matching complexity increases by supporting multiple granularities and many-to-many mappings, representing match results becomes a non-trivial problem. In addition, when a two-version matching tool is used for multi-version program analyses, aggregating individual matching results and representing the final results in a compact form remains as an open problem.

**Leveraging Dynamic Information.** Most matching techniques are based on syntactic similarities at a source code level. In comparison checking research [49, 39], dynamic information has been used to match an optimized version and an unoptimized version of the same program when the two versions were executed on the same input. Abstraction of multiple execution traces may guide matching of a static program representation. For example, comparing dynamic invariants [16] may be useful for identifying variable level matches at the entry (or exit) of a function.

# 8. CONCLUSION

In this paper, we defined the program element matching problem and argued its importance for fine-grained multi-version analyses. We presented a survey of matching techniques from various research areas and evaluated them based on hypothetical program change scenarios by hand. We believe that our assessment of existing techniques will guide researchers to choose an appropriate matching technique for their analysis.

In conclusion, every matching technique is an implementation of some pseudo equivalence function, and the more heuristics are used, the better the matching technique will work. One direction of future work involves building a hybrid matcher that leverages a feedback loop between matching tools and tools that infer refactoring events. Another future direction is to customize existing matchers in the context of a specific type of multi-version analysis and build an evaluation data set for that analysis. In addition, determining an optimal sampling interval for each matching technique remains as an open problem.

Our longer-term objectives are to (1) define the problem more precisely, allowing for better assessment and sharing of the approaches and (2) lay a foundation for more effective solutions applicable to specific kinds of multi-version analyses.

# 9. ACKNOWLEDGMENTS

We thank Dagstuhl 05261 seminar participants for fruitful discussions. We also thank Michael Toomim for reading our draft and Vibha Sazawal, Dan Grossman, and Rob DeLine for discussions that helped us refine our ideas.

# 10. REFERENCES

[1] subversion.tigris.org.
[2] www.cvshome.org.
[3] A. Aiken. A system for detecting software plagiarism.
[4] G. Antoniol, M. D. Penta, and E. Merlo. An automatic approach to identify class evolution discontinuities. In *IWPSE*, pages 31–40, 2004.
[5] T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *ASE*, pages 2–13. IEEE Computer Society, 2004.
[6] T. Apiwattanapong, A. Orso, and M. J. Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *ICSE*, pages 432–441, 2005.
[7] A. Apostolico and Z. Galil, editors. *Pattern matching algorithms*. Oxford University Press, UK, 1997.
[8] B. S. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, 24:49–57, 1992.
[9] I. D. Baxter, A. Yahin, L. M. de Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM*, pages 368–377, 1998.
[10] J. Bevan, J. E. James Whitehead, S. Kim, and M. Godfrey. Facilitating software evolution research with Kenyon. In *ESEC/FSE*, pages 177–186, 2005.
[11] J. Bevan and E. J. W. Jr. Identification of software instabilities. In *WCRE*, pages 134–145, 2003.

[12] D. Binkley, S. Horwitz, and T. Reps. Program integration for languages with procedure calls. *ACM TOSEM*, 4(1):3–35, 1995.

[13] J. R. Cordy. Comprehending reality: Practical barriers to industrial adoption of software maintenance automation. In *IWPC '03*, page 196, 2003.

[14] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *OOPSLA '00*, pages 166–177, 2000.

[15] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Trans. Softw. Eng.*, 27(1):1–12, 2001.

[16] M. D. Ernst. *Dynamically Discovering Likely Program Invariants*. Ph.D. Disseratation, University of Washington, Seattle, Washington, Aug. 2000.

[17] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *ICSM*, pages 190–197, 1998.

[18] C. Görg and P. Weißgerber. Error detection by refactoring reconstruction. In *MSR '05*, pages 29–35.

[19] C. Görg and P. Weißgerber. Detecting and visualizing refactorings from software archives. In *IWPC*, pages 205–214, 2005.

[20] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng.*, 26(7):653–661, 2000.

[21] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for Java software. In *OOPSLA '01*, pages 312–326, 2001.

[22] J. Henkel and A. Diwan. Catchup!: capturing and replaying refactorings to support API evolution. In *ICSE '05*, pages 274–283, 2005.

[23] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *PLDI'90*, volume 25, pages 234–245, June 1990.

[24] J. J. Hunt and W. F. Tichy. Extensible language-aware merging. In *ICSM*, pages 511–520, 2002.

[25] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Commun. ACM*, 20(5):350–353, 1977.

[26] D. Jackson and D. A. Ladd. Semantic Diff: A tool for summarizing the effects of modifications. In *ICSM '94*, pages 243–252, 1994.

[27] J. H. Johnson. Identifying redundancy in source code using fingerprints. In *CASCON*, pages 171–183. IBM Press, 1993.

[28] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.

[29] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy. An empirical study of code clone genealogies. In *ESEC/SIGSOFT FSE*, pages 187–196, 2005.

[30] S. Kim, K. Pan, and J. E. James Whitehead. When functions change their names: Automatic detection of origin relationships. In *WCRE*, 2005.

[31] S. Kim, E. J. Whitehead, and J. Bevan. Analysis of signature change patterns. In *MSR '05*, pages 64–68.

[32] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *SAS*, pages 40–56, 2001.

[33] J. Laski and W. Szermer. Identification of program modifications and its applications in software maintenance. In *ICSM*, 1992.

[34] E. Lippe and N. van Oosterom. Operation-based merging. In *SDE'92*, pages 78–87, 1992.

[35] G. Malpohl, J. J. Hunt, and W. F. Tichy. Renaming detection. *Autom. Softw. Eng.*, 10(2):183–202, 2000.

[36] T. Mens. A state-of-the-art survey on software merging. *IEEE Trans. Softw. Eng.*, 28(5):449–462, 2002.

[37] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *ICSE*, pages 284–292, 2005.

[38] I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. In *MSR'05*, pages 2–6.

[39] G. C. Necula. Translation validation for an optimizing compiler. In *PLDI '00*, pages 83–94, 2000.

[40] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *SIGSOFT '04/FSE-12*, pages 241–251, 2004.

[41] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[42] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM TOSEM*, 6(2):173–210, 1997.

[43] J. Sliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *MSR '05*, pages 24–28, 2005.

[44] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *ISSTA '02*, pages 97–106, 2002.

[45] W. F. Tichy. The string-to-string correction problem with block moves. *ACM Trans. Comput. Syst.*, 2(4):309–321, 1984.

[46] Z. Wang, K. Pierce, and S. McFarling. BMAT - a binary matching tool for stale profile propagation. *J. Instruction-Level Parallelism*, 2, 2000.

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

[48] A. T. T. Ying, G. C. Murphy, R. Ng, and M. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Softw. Eng.*, 30(9):574–586, 2004.

[49] X. Zhang and R. Gupta. Matching execution histories of program versions. In *ESEC/SIGSOFT FSE*, pages 197–206, 2005.

[50] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *MSR'04*, pages 2–6.

[51] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Trans. Softw. Eng.*, 31(6):429–445, 2005.

[52] L. Zou and M. W. Godfrey. Using origin analysis to detect merging and splitting of source code entities. *IEEE Trans. Softw. Eng.*, 31(2):166–181, 2005.