Renaming Detection

Guido Malpohl, James J. Hunt, Walter F. Tichy
University of Karlsruhe, IPD
Am Fasanengarten 5, 76128 Karlsruhe, Germany
{malpohl,jjh,tichy}@ira.uka.de

Abstract

Finding changed identifiers in programs is important for program comparison and merging. Comparing two versions of a program is complicated if renaming has occurred. Textual merging is highly unreliable if, in one version, identifiers were renamed, while in the other version, code using the old identifiers was added or modified.

A tool that automatically detects renamed identifiers between pairs of program modules is presented. The detector is part of a suite of intelligent differencing and merging programs that exploit the static semantics of programming languages. No special editor is needed for tracking changes. The core of the renaming detector is language independent.

The detector works with multiple file pairs, taking into account renamings that affect multiple files. Renaming detectors for Java and Scheme have been implemented. A case study is presented that demonstrates proof of concept. With renaming detection, a higher quality of program comparison and merging is achievable.

Keywords: merging, deltas, configuration management, static semantics

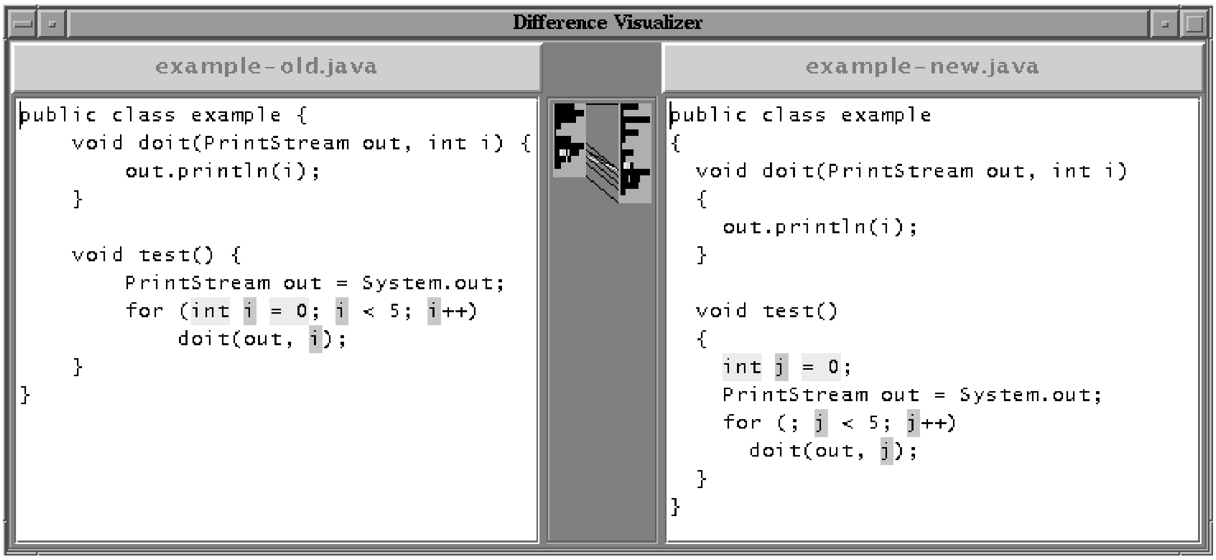
1 Introduction

Programmers often need to find out how files differ. Sometimes, changes have been made that have had unintended effects. On other occasions, someone else has modified code, during a programmer's absence. Perhaps, two files started out as identical copies, but were changed by different people and the changes need to be merged. This last case is especially frequent when geographically separate teams collaborate. No matter what the reason, automating the detection of changes can save a substantial amount of time.

The most effective programs today for these tasks are textual differencers and mergers. They are based on a classical dynamic programming algorithm by Hunt and McIllroy[7] dating back as far as 1976. This algorithm is still in active use today in numerous configuration management systems such as RCS[14], despite its well-known shortcomings. Textual differencers and mergers work on whole text lines. When a single character on a line is changed, the whole line is considered changed. Textual differencing thus misses a lot of information about the nature and extent of a change and confuse programmers with numerous false positives. For instance, if the formating of the program is altered, a textual differencer announces massive changes, even though there is no semantic difference. Similarly, if a code segment is moved, a textual differencer flags this segment as deleted in one place and notes an unrelated insertion of new code in another place. This disassociation leaves the task of finding the match to the programmer. Furthermore, a textual differencer cannot properly distinguish between changes in comments and program text. Also, if an identifier is consistently renamed, then a textual differencer cannot be instructed to ignore these differences, nor can this information be passed on to the merger for special treatment.

Textual mergers are based on textual differencers and thus share their problems; they also create new ones. For instance, if in one variant a call to a method was added whose name was changed in the other variant, the merger may produce a program that will not even compile; yet, the merger issues no warnings. Since most configuration management systems use this approach, programmers must rely on their compilers to find renaming conflicts.

This paper discusses an algorithm that solves the particularly difficult problem of detecting renamed identifiers. Its implementation, the *renaming detector*, is part of a suite of tools for differencing and merging that the authors are building. These tools avoid the drawbacks mentioned above, yet work with real programming languages and are fast enough for practical use. The tools are language-aware rather than textual: they are insensitive to formating, match moved code blocks properly, and treat comments and programming language statements differently. The *renaming detector* finds name changes that span multiple files, an important




```
Result:
From: (VAR:i|Primitive:int)|8)          To: (VAR: j|(Primitive:int)|10)
```

Figure 1. The difference visualizer with renaming detection

consideration for practical use. Its adaptation to multiple languages is achieved by providing a parser, an implementation for an abstract symbol analysis module, and perhaps some language specific rules for renaming analysis. Unlike other approaches, it does not require a special editor that maintains an abstract syntax tree and tracks the changes from one version to the next. Instead, it simply takes pairs of files as input.

Figure 1 shows an example use of the *renaming detector*, its output displayed with the *difference visualizer*. The left program is the base version, the right the variant. In the variant, the identifier i has been changed to j and the declaration moved out of the loop. Note that the layout of the variant has been drastically altered by pretty-printing, yet these changes are not flagged. The *difference visualizer* uses the *renaming detector*'s output to highlight the differences, making it easy for the programmer to focus on the relevant changes. Renamed identifiers are associated by color-coding (though this is not visible in the black and white reproduction).

The *renaming detector* first analyzes each file by itself, linking each declaration with its uses, i.e., with all references to the identifier introduced by the declaration. Next, it matches declarations in the two versions. It must do so by ignoring identifiers, since they may have changed. Instead, the matcher uses context information such as the types of variables and the similarity of the chains of references of each declaration. All declarations occurring in a program are matched; those that have different identifiers are returned.

While the *difference visualizer* and the underlying tools for detecting differences are operational, the merging program is still under construction. Renaming detection is especially interesting for merging, since it can be used for actively updating identifiers if one variant contains changed identifiers and the other changed code.

Prior work in merging includes the operation-based merging of Lippe et al.[8], the tree merge of Westfechtel[15] and the semantic merging of Reps et al.[3, 4, 1]. Lippe et al. require an editor that produces a re-playable history of transformations that change the base version into the variant. However, the authors do not indicate the level of these transformations nor where they come from. Keystroke logs of editors would certainly be too low-level. The author's approach can be seen as deriving these transformations, in particular the renaming transformations, given the base version and variant. Westfechtel defines merging as operations on abstract syntax trees, guaranteeing context-free correctness. Reps et al. aim at finding approximations for integrating the semantics, i.e., the behavior of program variants. The last two approaches require stable tagging of all program components (i.e, tree nodes in Westfechtel's system and assignments, loops, etc. in Reps et al's approach). Stable tagging means that the tags remain identical if the corresponding component was not changed. The tags are used to establish correspondence between versions. This requirement can be met with syntax-oriented editors that maintain the tags from one version to the next. Under these circumstances, renaming is easy to detect. A particular declaration of a variable, say, has the same tag in the base version and all its variants. Thus, renaming detection can be accomplished by a simple lookup.

The present approach does not assume the presence of tags or a specialized editor. The result of relaxing this constraint is a significantly more complicated matching problem, not only for matching changed identifiers, but also for matching program segments. This paper deals with the problem of detecting changed declarations, given only the base version and variant. It can be adapted to lexically scoped languages, primarily programming languages but also data description languages such as XML.

2 Design and Implementation

The basic idea of renaming detection is to examine two versions of a program for identifier definition and referencing and use a difference algorithm to relate deletions in the base version with insertions in the variant. The *renaming detector* has two goals that set additional requirements for the system. First, it must be able to work together with the *difference visualizer*: a tool being developed by the authors to present the difference between two versions of a system pictorially. Second, the system must also provide its results in a form suitable for merging. Since the entire system should be adaptable for use with any lexically scoped language, there is also the requirement that the entire suite of programs should be language aware, but not language dependent. In other words, as much language specific information as possible should be used without sacrificing adaptability to different programming languages. Since much of the processing involved in all three tasks is the same, the *renaming detector* should use as many components from the other tasks as possible.

The core components of the *difference visualizer*, a parser and a difference algorithm, are also core to renaming detection. Since the information gleaned from this process is to be displayed to the user, it is important to include all tokens in the difference in the same order as they appear in the original file. This means that, as opposed to normal parsers, grouping tokens such as `BEGIN` and `END` in Pascal or `{` and `}` in **Java**, can not be discarded. It also means that the usual way an operator is stored in the parse tree as an internal node with its arguments as leafs is undesirable. When serializing such a parse tree for display, token ordering is dependent on the type of node being processed. For example, in some languages, some operators are written in prefix notation and other nodes are written in infix notation; still other languages use postfix notation. In order to avoid having to examine every node during serialization, it is easier to put all tokens on leaf nodes in the order they occur in the source code and add special internal nodes to structure them. This also makes it easier to keep the difference algorithm's view of continuous tokens the same as what the user will be shown on the screen. These requirements taken together result in a parse tree where the original token stream can be reconstructed by visiting all leaf nodes depth first.

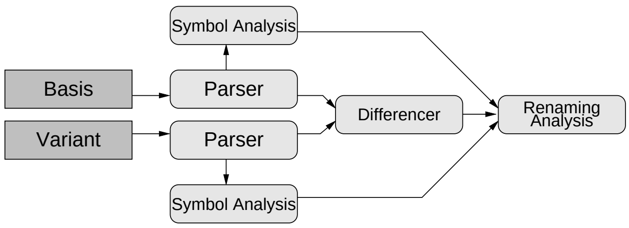


Figure 2. *Renaming Detector* **Overview**

The merge algorithm will use the core components of the difference algorithm, as well as the results of the *renaming detector*. In addition, it also needs to use identifier definition and reference information for conflict resolution. This is the same symbol analysis that the *renaming detector* needs for finding renamed identifiers. Thus symbol analysis should be structured such that it can be used by both subsystems.

The overall architecture of the *renaming detector* consists of a parser, a differencer, symbol analysis, and renaming analysis. The parser provides the basis for the differencer and for symbol analysis. Then, the information derived from symbol analysis and differencing feeds renaming analysis. The latter associates identifiers in the base version with identifiers in the variant on the basis of similarity of their use without regard to their names.

2.1 A Parser for All Programs

Aside from the shape of the resultant parse tree, parsing for renaming is no different than for any other type of program analysis. Since the system should work for any given lexically scoped language, it makes sense to use a parser generator instead of writing a new parser for each language. The authors have chosen **Antlr**[12, 11] for this task for three reasons. **Antlr** is an $LL(k)$ parser with predicates which makes writing new grammars relatively easy, since the input language is an extended BNF format. **Antlr** provides parser channels, which make it easy to handle comments. Finally, it generates **Java** code. **Antlr**, like most parser generators, does not dictate the shape of the resultant parse tree, so there is no problem giving the resultant parse tree the form required by the *difference visualizer*'s difference algorithm.

2.2 Definitions and References

Once the files have been parsed, symbol analysis can be used to catalog all definitions and their references. Basically, this is the same symbol table generation and analysis that one does in a compiler. Of course, the details are different, because the shape and content of the parse tree is different. A significant difference is that named blocks such as subprograms, methods, classes, and records are tracked for comparing the bodies of identifier definitions during renaming analysis.

2.3 An Adaptive Approach to Differencing

Calculating the difference between the two files is a the heart of renaming recognition. A good differencer is necessary to establish the mapping between each file pair. From the prospective of the rest of the system, it is not so important which algorithm is used, so long as certain key features are provided: it must be fast, operate at a token level, find moved sequences, and respect the structure of the program being analyzed. The authors have chosen a new adaptive heuristic for tree comparison[5] for this task.

It would seem appropriate to use current tree comparison algorithms. However, making a full tree comparison is too slow for practical application. By using a sequence differencer over the tokens of a program and only using the parse tree information where more than one possible match is found, one can do much better. A further trick is not to throw out the seemingly insignificant tokens such as grouping and separation tokens, e.g. curly braces and semicolons in **Java**, during parsing. It is precisely these tokens that the parser uses to build the tree structure in the first place. By keeping them in the token stream, tree structure information is implicitly used in the comparison. As mentioned above, this approach also has the advantage of making the result of the comparison easier to interpret by a user.

The differencer essentially composes the variant out of segments of tokens from the base version. The result is a sequence of base segments that piece together the variant, with some insertions in the case that the base does not contain certain variant tokens. The sequence of base segments must meet two requirements:

Coverage— every token in the variant that has a match in the base must be included in a segment, i.e. insertions must be minimized; and

Minimal Sequence—the number of segments in the sequence must be minimal, so as to avoid fragmentation into many small segments.

Further, when several equally good matches are available for any given block (sequence of tokens), the algorithm uses the longest weighted upsequence to chose among those blocks. An upsequence is an ordered list of segments that have increasing start addresses in the base and do not overlap. The weight of each segment is its length. Thus the sum of the lengths of the segments that belong to the upsequence should be maximal. Segments that form an upsequence have the same order in base and variant. Maximizing the upsequence means that there is a maximal "skeleton" common among base and variant. The skeleton makes it easy for a user to view the differences, because it minimizes the number of segments whose locations cross from the base to variant. It is also beneficial for renaming detection, because it tends to keep base sequences that were split up by modifications in the original order.

McCreight's suffix tree algorithm[9] appears to be an ideal basis for a fast token sequence comparator. In fact, it has already been used to implement effective delta compressors[6]. However, it has one major drawback for sequence comparison: only the first best match is stored in each node of the suffix tree. Finding all others requires an expensive search. Just being able to find the first best match is not sufficient, because the first best match is not always the one that fits into the longest upsequence.

To improve the precision of the matching, one can save references to all best matches that have been found. This involves having a list of the start position of all matches represented by each node. Since the number of nodes visited in the tree does not increase, this change does not disturb the linear runtime of the suffix algorithm, neither during creation nor lookup.

After building the suffix tree over the base version, the match process consists of scanning the variant, using the suffix tree to look up token sequences along the way. This is essentially the process used by Tichy in his **bdiff** delta compressor[13]; except, whereas his algorithm obtained the first best match from the suffix tree, this modified tree returns a list of all best matches. Thus an additional processing step is needed to chose which of the best matches to use for forming the longest upsequence.

The solution to this problem was inspired by Ousterhout's corner stitching data structure[10]. Just as in the classic longest common subsequence algorithm, the correspondence between two files can be depicted as a plane, where each file is an axis of the plane. The difference is that the modified suffix tree algorithm has already computed the best possible matches. These matches are represented as rectangles in the plane. Finding the best correspondence between two sequences amounts to searching the diagonal of the plane. Using corner stitches yields a compact representation of the matches, while making traversing the diagonal easy. In practice, Ousterhout's algorithm needs to be modified extensively for this task, but the essential motivation is the same.

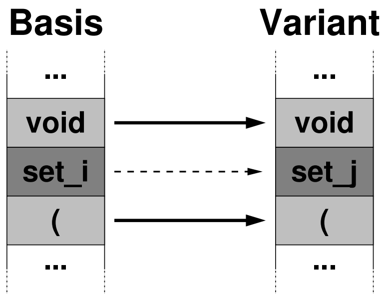


Figure 3. Declaration Similarity

There are two final complications to the algorithm. The first is that segments can overlap in the base version. This is where syntax tree information is used to decide how to break the overlap. The second is that in certain cases, a recursive matching step is performed to fill out the longest upsequence.

The performance of this algorithm has a linear lower bound on the length of the two token sequences. The suffix tree can be built in linear time of the length of the base sequence and the matching is linear in the length of the variant. Building the stitched plane requires examining all matched blocks. When each block has only one match, this process is linear; but in the worst case, the runtime is proportional to some fraction of the length of the base times the length of the variant. The entire algorithm has an upper bound of the square of the average length of the token sequences.

A complete description of the algorithm exceeds the scope of this article. The main point is that this algorithm has the essential characteristics of being fast, operating on a token level, finding moved sequences, and maintaining the structure of the program being analyzed. More information can be found in a technical report[5].

2.4 Renaming Analysis

Now all the raw materials are available for finding renamed variables. Information from the difference algorithm is used to map tokens between the base version and the variant without regard to name equivalence. For each definition (i.e., a language construct that introduces a new identifier), the program uses this mapping to assemble facts about declaration, implementation, and reference similarity between identifiers in the base version and the variant. These facts are then used by an expert system to find renamed identifiers.

Declaration similarity is defined by the tokens surrounding the appearance of the name at the place where it is introduced in the definition (typically at the beginning). The neighboring tokens in the base are examined to see if they map together to the same relative place in the variant. If they both map to the same relative place, i.e. one before and one after, then a possible match is registered. Figure 3 illustrates this for method **set_i**. This is the weakest form of similarity.

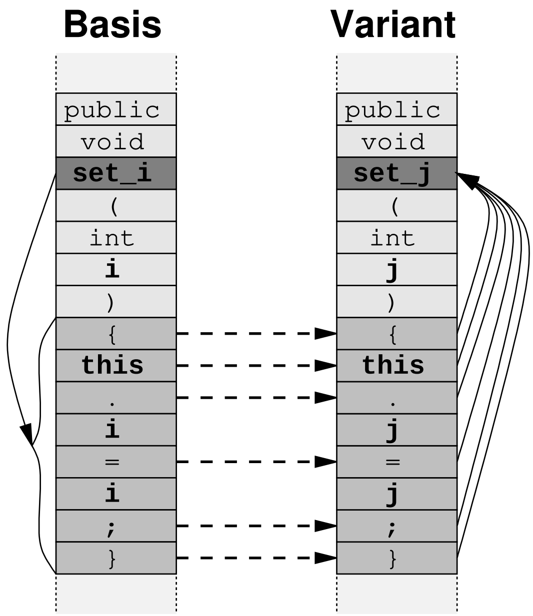


Figure 4. Implementation Similarity

Implementation similarity is defined by mapping the tokens in the body of a definition with tokens in the bodies of definitions in the variant. A percentage of sameness is calculated for each body where at least one mapping is found. This percentage is calculated as the number of matched tokens from the base body in the given variant body relative to the average size of the two bodies. This gives possible matches with associated likelihood. Figure 4 depicts an 75% correspondence between method **set_i** in the base and method **set_j** in the variant. This similarity is stronger than declaration similarity. However, its strength is dependent on the body length.

Reference similarity is the most complicated to calculate, but is also the strongest kind of similarity. It is essentially a combination of the techniques used for declaration and implementation similarity. For each definition in the base version all references are examined. For each reference of the definition in the base, neighbor tokens are used to map to a corresponding reference in the variant. As in declaration similarity, the neighboring tokens in the base are examined to see if they map together to the same relative place in the variant. If they both map to the same relative place, a match is registered for its definition in the variant. Once all the references have been processed, a percentage

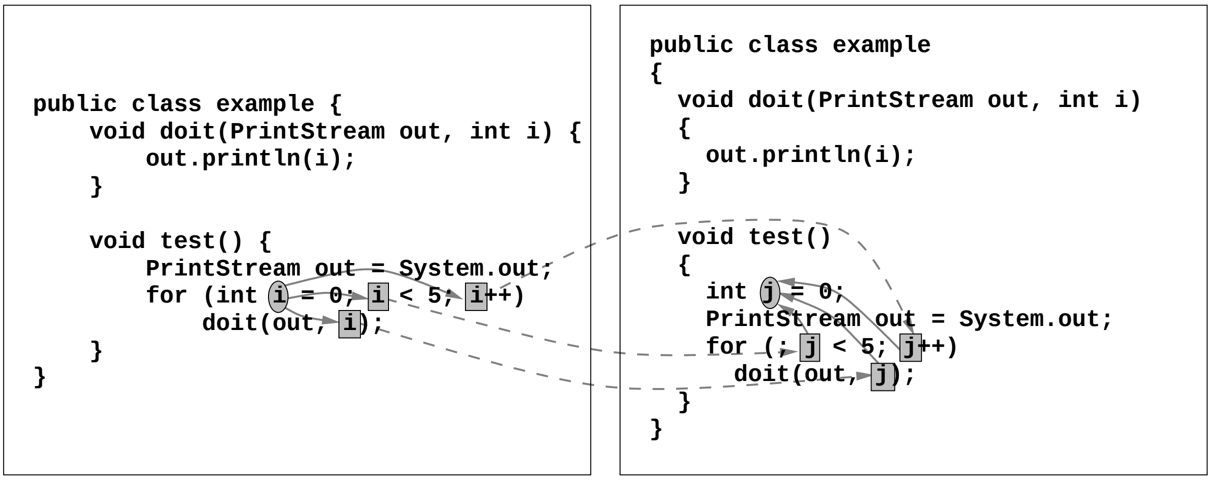


Figure 5. Reference Similarity

of sameness is calculated for each definition in the variant for which at least one match has been found. As with implementation similarity, this gives possible matches with associated likelihood. Figure 5 depicts a perfect match, where all three references of **i** map to references of **j**, all of which have the same declaration.

The final stage is to bring all the facts together to determine the final mapping between identifiers in the base and the variant. The expert system **Jess**[2] is used to weigh the relevance and extent of each type similarity for each possible pair of identifiers to find the pairs that correspond to one another. In general, using an expert system instead of a hard coded search algorithm gives the system flexibility for further experimentation. Specifically, there is room for possible improvements in the way neighboring tokens are used to aid mapping.

2.5 From the Trees to the Forest

Renamings in real projects often cross file borders. The naive way to handle this case is just to parse all files in a project for both the base version and the variant and then used the two sets of resulting parse trees for further analysis. Unfortunately, this would exceed the resources of most available machines for any reasonably large project. Instead, the process is broken down into multiple passes. Intermediate files can then be used to share important information between file pairs.

Two types of information are needed for global analysis: information about externally visible definitions and about external references, i.e. those that are not resolvable in the same file where they occur. In order to analyze external references, information is needed about externally visible definitions. The final renaming determination requires information about these external reference. This means that, in order to process files pairwise, one must use three passes.

The current implementation does precisely this. The first pass just parses the individual files and writes out summary information about externally visible definitions. The second re-parses the files and performs symbol analysis, differencing, and analysis of references of externally defined identifiers to record the external references. The last recapitulates the first two passes, except that references to internal definitions are examined instead of to external ones, and then both are combined in the final renaming analysis. The latter two passes work on file pairs provided by a configuration management system, hence it is up to that system to recognize file renaming.

A better implementation would save intermediate data to eliminate repeating work from previous passes. Problems with **Java** serialization prevented the authors form a timely implementation of this approach. This change has been left for future work. Though the system works as is, one could expect about a doubling in performance if this change is made.

2.6 Babylon Revisited

A major design goal for the system was to minimize the size and number of the language dependent modules and to define clean interfaces for them. For every language, three modules have to be implemented: a parser, a symbol analysis, and a small part of the renaming detection itself. This design was developed for the Java front-end and proved to be appropriate when adding the Scheme support later.

When using a parser generator, implementing a parser usually consists of just converting the language specification into the parser generator language. The major part of

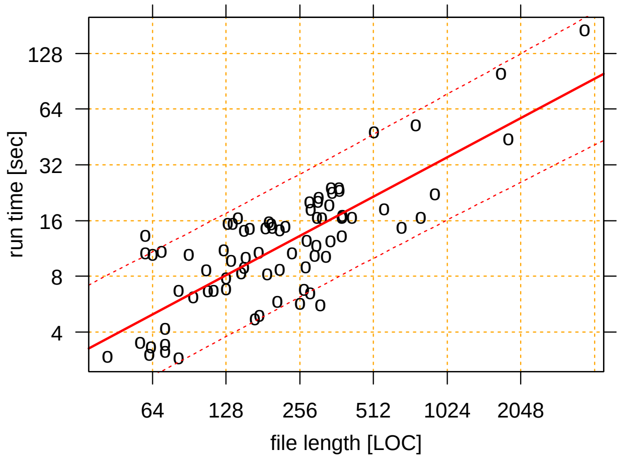


Figure 6. Time needed to compare a pair of programs. *File length [LOC] is the sum of both line counts. The linear regression is displayed with the 90% quantile (stddev = 8.275 and r = 0.9276)*

the effort is required for the symbol analysis. Depending on the complexity of the reference resolution, implementation can take anywhere from four days to several weeks. Surprisingly little work has to be done for the renaming detection module. It can be done in about 300 LOC. An initial pass can be completed in a matter of hours, but tuning may take much longer and should be regarded as open-ended. Nevertheless, the algorithm starts to show good results after a few hours.

For Scheme, approximately two working days were needed to write the parser. The development of the symbol analysis took five days. The language dependent parts for Scheme consists of a rule file (approx. 270 LOC) and a small Java class (less than 30 LOC). The implementation of this module required only a few hours to write. Overall, the time needed to implement a front-end for a "simple" language like Scheme is around one and a half weeks.

3 A Case Study

This section describes the real-world performance of the system in regard to run time and detection quality. The authors selected two successive configurations of the *renaming detector* itself for study. As developers, they had prior knowledge about the renamings that occurred and hence were able to verify the results of the detector easily. The package consists of 77 Java files, ranging from 21 to 1863 lines of code with an average of 166 LOC. The total size of the second configuration is 12804 LOC or 379 KB.

The difference between the two configurations is the introduction of a generalization interface, which led to changes in 34 files. Two kinds of changes were made. There were 49 occurences of variable type changes (local, instance, or class). In 37 cases, these were accompanied by a change of the name of the variable. The signatures of methods (parameter types and return types) were altered in 46 cases, whereby 10 of the methods were also renamed. There were no other changes except for alterations in comments and import statements. In sum, there were 47 renamings in the source code.

Not only the total running time of the algorithm was measured, but also the time needed to compare every single pair of files. The measurements were carried out on an Intel PIII-450 CPU with 128MB RAM using Linux 2.2 and JDK1.2 with the Just-In-Time compiler disabled.

The total processing time for 77 file pairs is 1285 seconds. Note that unchanged files are included, since the information about external references must be extracted from these files. Although the total run time is fairly long, the average time needed for one file is only 16.7 seconds. Of the three phases, the first one needs 7%, the second 42%, and the third 51%. Nevertheless, little time is spent in the actual renaming detection. Computation time appears to be consumed mostly by the reference resolution algorithms.

The time needed to compare a pair of files (over all three phases) is dependent on the size of the files. Figure 6 shows the run time relative to the total size of a file pair in lines. Despite the theoretically possible quadratic behavior, practical run times are roughly proportional to file sizes, with a correlation coefficient of $r = 0.93$.

For rating detection performance, all files were analyzed beforehand and the 47 renamings recorded. The automated system was able to locate 40 of these. No false positives were flagged. Altogether 2296 declarations had to be paired among the two configurations. The system failed in 12 cases to make a connection; 7 of these are the missing renamings. The reasons the system failed to associate the declarations were mostly missing references or references which could not be associated, since their neighborhood had changed too much. Altogether the matching ratio is 99.6% with 100% precision (no false matches). The renaming detection rate is $40/47 = 85\%$, also with 100% precision.

The authors believe the performance on this benchmark to be quite impressive, although improvements in both speed and detection rate appear possible. For instance, using compiled Java and a faster analysis of unchanged files would improve performance drastically. Analysis of the missed renamings also suggests improvements of the matching algorithms.

4 Conclusions

The authors have developed algorithms for detecting changed identifiers in lexically scoped programming languages without the need for special editors. The method can also be adapted to data description languages such as XML. Renaming detectors for both Java and Scheme have been built. A case study demonstrates practicability and scalability of presented methodology. The next step is to improve the matching rate without sacrificing precision, tuning the algorithms for speed, and developing a renaming-cognizant merger.

References

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

[2] E. J. Friedman-Hill. *Jess, The Java Expert System Shell*, June 1999.

[3] S. Horwitz, J. Prins, and T. Reps. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, 1989.

[4] S. Horwitz, T. Reps, and W. Yang. A program integration algorithm that accomodates semantics-preserving transformations. *ACM Transactions on Software Engineering and Methology*, 1(3):310–354, 1992.

[5] J. J. Hunt. An adaptive heuristic for tree comparision. Technical Report 2000-9, Fakultät für Informatik, Universität Karlsruhe, 2000.

[6] J. J. Hunt, K.-P. Vo, and W. F. Tichy. Delta algorithms: An empirical analysis. *ACM Transactions on Software Engineering and Methodology*, 7(2):192–214, Apr. 1998.

[7] J. W. Hunt and M. D. McIllroy. An algorith for differential file comparison. Computer Science Technical Report 41, Bell Laboratories, June 1976.

[8] E. Lippe and N. van Oosterom. Operation-based Merging. In H. Weber, editor, *Proceedings 5th ACM SIGSOFT Symposium on Software development environments*, pages 78–87, 9–11 December 1992.

[9] E. M. McCreight. A space economical suffix tree construction algorithm. *Journal of the ACM*, 32:262–272, 1976.

[10] J. K. Ousterhout. Corner stitching: a data structuring technique for VLSI layout tools. Technical Report CSD-83-114, University of California, Berkeley, 1983.

[11] T. Parr. *ANTLR Reference Manual*, May 1999.

[12] T. Parr and R. Quong. A Predicated-LL(k) parser generator. *Software Practice and Experience*, 25(7):789–810, July 1995.

[13] W. F. Tichy. The string-to-string correction problem with block moves. *ACM Transactions on Computer Systems*, 2(4):309–654, November 1984.

[14] W. F. Tichy. RCS – a system for version control. *Software – Practice and Experience*, 15(7):637–654, July 1985.

[15] B. Westfechtel. Structure-Oriented Merging of Revisions of Software Documents. In S. Feiler, editor, *Proceedings 3rd International Workshop on Software Configuration Management*, pages 68–80, New York, 1991. ACM Press.