Bug Classification Using Program Slicing Metrics

Kai Pan, Sunghun Kim, E. James Whitehead, Jr.
Dept. of Computer Science
University of California, Santa Cruz
Santa Cruz, CA 95064 USA
{pankai, hunkim, ejw }@cs.ucsc.edu

Abstract

In this paper, we introduce 13 program slicing metrics for C language programs. These metrics use program slice information to measure the size, complexity, coupling, and cohesion properties of programs. Compared with traditional code metrics based on code statements or code structure, program slicing metrics involve measures for program behaviors. To evaluate the program slicing metrics, we compare them with the Understand for C++ suite of metrics, a set of widely-used traditional code metrics, in a series of bug classification experiments. We used the program slicing and the Understand for C++ metrics computed for 887 revisions of the Apache HTTP project and 76 revisions of the Latex2rtf project to classify source code files or functions as either buggy or bug-free. We then compared their classification prediction accuracy. Program slicing metrics have slightly better performance than the Understand for C++ metrics in classifying buggy/bug-free source code. Program slicing metrics have an overall 82.6% (Apache) and 92% (Latex2rtf) accuracy at the file level, better than the Understand for C++ metrics with an overall 80.4% (Apache) and 88% (Latex2rtf) accuracy. The experiments illustrate that the program slicing metrics have at least the same bug classification performance as the Understand for C++ metrics.

1. Introduction

Over time, an active software system keeps growing in size and complexity, and gets increasingly hard to understand and maintain. Software engineers use code metrics as one mechanism for quantitatively measuring software qualities, thereby helping them better understand and maintain software systems.

There are many code metrics that indicate the design or source code properties of programs, such as size, complexity, coupling, cohesion, etc. To measure these properties, most code metrics focus on 'static' syntactic aspects of program components, such as lines of code (LOC), number of declarations, number of functions, function fan-in, function fan-out, etc. The cyclomatic complexity [1] metric is somewhat more complex, as it uses program control flow to determine program complexity. To capture more fine-grained program properties, we introduce a set of source code metrics for C programs using program slicing information. That is, the program slicing metrics measure the size, complexity, coupling, and cohesion properties of programs based on program slices, vertices in slices, and dependence edges in slices.

Program slicing techniques [2, 3] study the behavior of source code through the flow dependency and control dependency relationships among statements. A program slice consists of all the statements that may influence the values of a variable at a program point, and a program or program slice can be represented by a program dependence graph [3, 4]. We compute program slicing (PS) metrics for C programs based on intra-procedural slices with respect to output variables in every function. By using program slicing information, PS metrics capture more fine-grained program properties related to program behaviors.

To evaluate PS metrics, we compare them with traditional code metrics computed using the Understand for C++ tool [5] by performing a bug classification experiment. In this experiment, we predict whether a source code file or a function is buggy or bug-free, based on the information available in the metrics. It is intuitive to think that software properties such as size, complexity, and coupling are correlated with software bugs, and several research efforts provide evidence of this correlation [6-8]. In our experiment, we compute the average summary values of PS metrics and Understand for C++ (UC) metrics for 76 revisions of the Latex2rtf project and 887 revisions of the Apache HTTP version 1.3 project. These project revisions are extracted from their SCM repositories using the Kenyon infrastructure [9]. The PS metrics and UC metrics are used to predict which files or functions will have bugs. To determine the predictive accuracy, we collect the actual bugs existing in the same revisions of Latex2rtf and Apache, then compare

them against the predicted results. The experiment shows that the precision achieved using PS metrics is slightly better than that for UC metrics.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 provides a detailed description of the metrics comprising the UC and PS metric sets. Section 4 describes our bug classification experiments using UC metrics and PS metrics respectively, and analyzes the results. Section 5 discusses tradeoffs when using PS metrics and we conclude in Section 6.

2. Related work

Program behavioral properties have been considered when computing program metrics in several research efforts. Cyclomatic complexity is a well-known software metric that indicates the logical complexity of a program [1]. To compute cyclomatic complexity, the program control flow is examined to measure the number of independent paths in the source code. In [2], Weiser first suggested the potential use of program slicing techniques to measure programs. In [10], Ott and Thuss introduced two new metrics for code cohension, *MinCoverage* and *MaxCoverage*, supplementing the existing cohesion metric introduced by Weiser. Bieman and Ott [11] introduced a method that measures functional cohesion using program slicing information, in which the number of tokens that are shared by multiple slices are used to represent cohesion. Li [12] explored the use of programming in measuring coupling in object-oriented programs. Work in [13] empirically investigated the use of cohesion metrics to identify degraded modules and guide software reconstruction.

Software metrics are widely used to measure software quality or predict bug-prone parts of software. Gyimothy et al. identify object-oriented metrics and use them for fault prediction [6]. Khoshgoftaar and Allen have proposed a model to list modules according to software quality factors such as future fault density [7, 8]. The inputs to the model are software complexity metrics such as LOC, number of unique operators, and cyclomatic complexity. A step-wise multi regression is then performed to find weights for each factor. Rather than determining a good bug prediction model using existing metrics, we develop several new metrics using program slicing, and test if the new metrics can predict bugs better than conventional metrics (UC metrics [5]), using a Bayesian network classifier instead of a regression model.

3. UC metrics and PS metrics

3.1. UC metrics

The UC metrics are generated by the Understand for C++ tool [5], which computes most traditional code metrics for C and C++ programs. The UC metrics set contains 46 metrics, which are generally categorized into three groups: project-level, file-level, and function-level. We only use the file-level and function-level UC metrics for C in our experiment. We describe several major function-level metrics in the UC set below. Most file-level UC metrics are sums or averages of the metrics for the functions in the file.

CountLine. The number of lines in the function.

CountLineCode. The number of lines that contain source code.

CountLineComment. The number of lines that contain comments.

CountStmtExe. The number of executable statements.

CountInput. The number of inputs used by a function (Inputs include parameters and global variables that are used in the function.)

CountOutput. The number of outputs, parameters or global variables that are set in a function.

CountStmtDecl. The number of declaration statements in a function.

Cyclomatic. The cyclomatic complexity of a function.

MaxNesting. The maximum nesting of control statements in a function.

In addition to the metrics listed above, the UC metrics set contains the metrics *CountLineBlank*, *CountLineDecl*, *Count-LineExe*, *CountLineInactive*, *Cyclomatic-Modified*, *CyclomaticStrict*, *CyclomaticMax*, *CyclomaticMax-Modified*, and *PercentComment*, whose definitions can be found in [5].

3.2. Program Slices and Dependence Graph

Before defining our program slicing based metrics, we first provide some background on program slicing and dependence graphs.

In the program slicing perspective, a function may contain multiple behavioral aspects, such as all the statements that change the value of a global variable, or statements that compute the return value of the function. We use program slices to capture behavioral aspects of a function. A function contains one or more intra-procedural programs slices, each of which is with respect to the output variable of this function. An output variable of a function can be the function's return value or a non-local variable modified in the function.

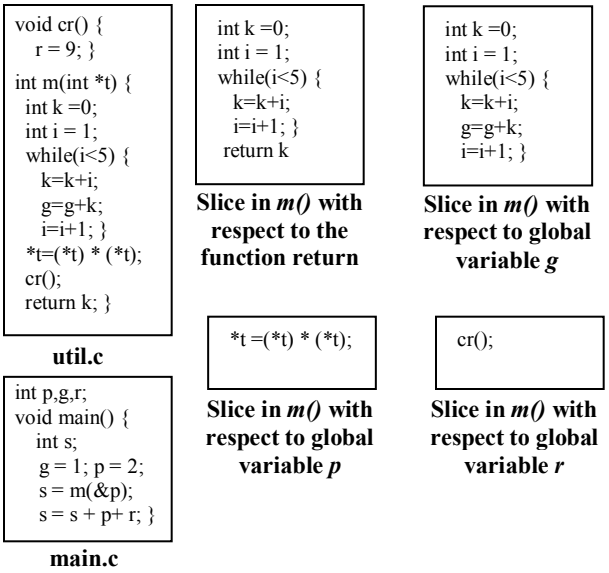

Figure 1. An example program and the slices in the function m()

Figure 1 shows an example program, which consists of *util.c* and *main.c*, and four program slices in the function *m()*, which are the slices with respect to the function's return value, and the global variables *g*, *p*, and *r* respectively. The four slices in function *m()* mean that the function *m()* does four 'things': it computes the return value (1), and modifies the value of global variables *g*, *p*, and *r* (2-4). Note that in the slice with respect to *p*, *p* is modified by dereferencing the pointer variable *t*; and, in the slice with respect to *r*, *r* is modified indirectly through a function call.

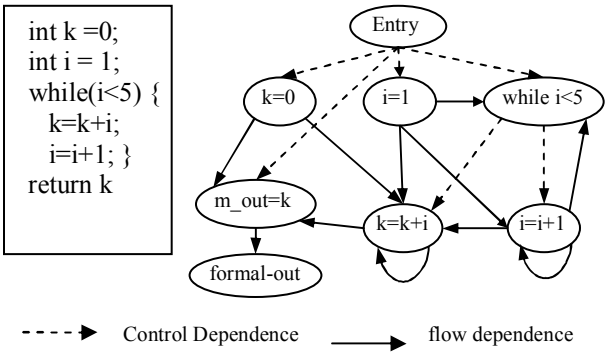

Figure 2. PDG for the slice with respect to the function return of function m().

Program slices can be represented by program dependence graphs. A program dependence graph (PDG) is a directed graph, where each vertex in the PDG represents a statement or predicate in the program, and each edge represents a data dependence or control dependence relationship between two vertices. Figure 2 shows the PDG for the slice with respect to the function return in function *m()* as presented in Figure 1. In Figure 2, *m_out* is a temporary variable that stores the return value of function *m()*. When constructing the PDG for a function, there is a formal-out vertex in the PDG if the function has a non-void return value, and there is a global-formal-out vertex for a global variable or a group of global variables that may be modified in the function. In our PS metrics implementation, we have to compute the PDG for function slices, since the vertices and edges are factors used in computing PS metrics.

3.3. PS metrics

Like UC metrics, PS metrics indicate the size, complexity, coupling and cohesion of C programs. Unlike UC metrics, PS metrics are based on program slice information, which is of finer-granularity than the measures in UC metrics. Program slices have the additional advantage of capturing program behavior, and hence metrics based on slices are more directly related to these behaviors. We explain each of the PS metric items as follows.

sliceCount. The number of slices a function contains. *SliceCount* is similar to *CountOutput* in the UC metrics, but *sliceCount* considers the global variables modified through the dereferencing of pointer variables. To achieve this, pointer analysis is used in the implementation. In the example of Figure 2, the *sliceCount* for function *m()* is 4.

verticesCount. The number of vertices in a function's program dependence graph. This metric is similar to *CountLineCode* in the UC metrics, but *verticesCount* is more fine-grained. For example, it treats the statement *'a = b()'* as two statements: one function call and one assignment. It additionally counts implicit vertices, such as global-formal-out vertices.

edgesCount. The number of dependence edges in a function's program dependence graph. In contrast to the *verticesCount* metric, which weights each statement evenly, *edgesCount* is based on the control or flow dependence relationships of each statement with other statements. For example, the *while* statement in Figure 1 has three outgoing control dependence edges to the three statements it encloses, so it contributes 3 to the *edgesCount* metric, while the statement *'g=g+k;'* only contributes 1 flow dependence edge, which is from the vertex representing this statement to the global-formal-out vertex for *g*. This metric represents a combination of a function's size and logical complexity.

edgesToVerticesRatio. For a given function's program dependence graph, the ratio of the number of dependence edges to the number of vertices. This metric indicates how much the statements in a function depend on each other by control or data flow. A high *edgesToVerticesRatio* indicates more logically complex code.

sliceVerticesSum. The sum of the vertices contained in each slice in a function. This metric is a combination of slice count and slice size.

maxSliceVertices. The number of vertices of the slice that have the maximum vertices in all the slices of a function. A count of the vertices in the one slice that contains the most vertices of all slices in the function.

globalInput. The number of function parameters and non-local variables used in a function. This metric is similar to the UC metric *CountInput*, but *globalInput* additionally considers the non-local variables introduced by pointers.

globalOutput. The number of non-local variables modified in a function. This metric is similar to the UC metric *CountOutput*, but *globalOutput* also considers the non-local variables introduced by pointers.

directFanIn. The sum of function slices in the other module files that use the output variables directly modified in this function. In the example of Figure 1, the *directFanIn* metric for *m()* of *util.c* is 3, since the return value of *m()*, and the global variables *p* and *g*, are used in function *main()* of *main.c*. The global variable *r* is not counted for the *directinFanIn* of *m()*, since *r* is not directly modified in *m()*, but indirectly modified in *cr()*. This metric is a combination of function fan-in and function outputs and is a measure of the coupling of the function.

indirectFanIn. The sum of slices in other module files that use the output variables indirectly modified in this function.

directFanOut. The sum of function slices in the other module files whose output variables are directly modified in them and used in this function. In the example of Figure 1, the *directFanOut* metric for *main()* of *main.c* is 3, since the return value of *m()*, and the global variables *p* and *g* are used in *main()*.

indirectFanOut. The sum of function slices in the other module files whose output variables are indirectly modified in them and used in this function.

lackOfCohesion. This metric indicates the cohesion of function slices. Like the method in [11], we determine *lackOfCohesion* by the overlap ratio of function slices, i.e. how much the slices in a function share the same vertices. But, since it is expensive to compute the ratio of shared vertices among a number of function slices, we calculate an approximate ratio of slice overlap by computing the ratio of the program lines each slice covers to the total number of lines in the function. Consider function *m()* in Figure 1. The number of lines in the function body of *m()* is 9, while the four slices in *m()* have 6, 6, 1, and 1 lines respectively. Hence, the slice overlap ratio for function *m()* is (6/9+6/9+1/9+1/9)/4, or 39%. The *lackOfCohesion* value is the reciprocal of the slice overlap ratio. So, for function *m()*, its *lackOfCohesion* value is 2.56. We can see from Figure 1 that the slice with respect to the function return has high cohesion with the slice with respect to *g*, while they both have low cohesion with the slice with respect to *p* or *r*.

The PS metrics listed above are computed at the function level of granularity. We have also developed a set of metrics at the file level corresponding to those at function level. The file level metrics are the sum of the function-level metrics for all the functions a file contains, except for *lackOfCohesion* and *maxSliceVertices*, which are the average of the function-level metrics for all the functions in the file, and *edgesToVerticesRatio*, which is the ratio of sum of edges in a file to the sum of vertices in the file. Note that, in practical computation of the program slices for functions, some normalization will be performed for complex statements. So, the actual PS metrics will take into account additional vertices and edges generated by normalization.

4. Experiment

It makes intuitive sense that code properties like size, complexity, and coupling are correlated with bug-proneness, and several research efforts have demonstrated this correlation. Ideally we would like to demonstrate that the PS metrics are also correlated with bug-proneness, and perform better than the traditional metrics in the UC set. In order to evaluate the PS metrics, we compared the PS metrics and UC metrics by applying both of them to the task of bug classification, in which we use machine learning techniques to predict whether a source code file or function will contain bugs (or not) based on the correlation of code metrics with facts gathered from actual buggy/bug-free code.

We performed an experimental bug classification on 887 revisions of the Apache HTTP server project and 76 revisions of the Latex2rtf document converter project using PS metrics and UC metrics respectively. The basic data for the two projects is described in Table 1. Both file-level and function-level PS metrics and UC metrics are used in the experiment. We additionally compared the accuracy of the PS and UC metrics to evaluate the PS metrics.

4.1. Overview of Experimental Process

We use the Latex2rtf project as an example while explaining the steps used to perform bug classification at the file and function level. The same process is used for the Apache HTTP project.

Step 1: Fact Extraction and Metrics Computation. We compute the PS and UC metrics at both the file and function level for all files in every revision. After all revisions have been processed, we compute the average summary value set of the PS metrics and UC metrics for the source code files and functions in the Latex2rtf project. The metrics value set consists 5 values for each

metric, which are the average metric value of the revisions examined, the maximum metric value in the revisions, the standard deviation of metric values, the cumulative value difference of the metric value along the revisions, and the change tendency (-1, 0, 1) of metric values in the revisions examined.

Step 2: Bug Labeling. The change logs of the 76 revisions of the Latex2rtf project are mined to determine bug-introducing changes [14] (changes to the source code that introduce an error in the code). We mark the files and functions that have ever contained bug-introducing changes as buggy ones, and mark the others as bug-free.

Step 3: Bug classification. We use the 10-fold cross-validation method [15] to perform bug classification prediction using the PS metrics value set from step 1. We compare the classification prediction with the real bug classification from step 2 to obtain the predictive accuracy of the PS metrics. We follow the same steps using the UC metrics to obtain their predictive accuracy.

Step 4: Accuracy Comparison. We evaluate the PS metrics by comparing the classification accuracy from PS metrics with the classification accuracy from the UC metrics at the file level and the function level respectively.

We explain each of these steps in further detail in the following subsections.

4.2. Fact Extraction and Metric Computation

To compute the PS and UC metrics for the Latex2rtf project, we retrieved 76 revisions of the Latex2rtf project source code from its CVS repository [16] using Kenyon [9]. Kenyon automates the process of extracting revisions from a software configuration management system repository, computing user-specified metrics on each extracted revision and then storing them in a database.

For each retrieved revision we use the Understand for C++ tool to compute the UC metrics at the function and file level. After all revisions are processed, we obtain the UC metrics value set for each file and function.

We compute the PS metrics as follows. For each revision retrieved, we use CoderSurfer [17] to perform static analysis on each file. CodeSurfer is a program analysis tool produced by GrammaTech, Inc., capable of parsing and analyzing C programs, generating program dependence graphs, and performing program slicing for C programs. We computed the function-level PS metrics for each function based on its procedure dependence graph generated by CoderSurfer. The file-level PS metrics are obtained after all functions in a file are analyzed. Once all 76 revisions of the Latex2rtf project are processed, the PS metrics value set for each function and file are computed. An an example, Table 2 shows the average metric value of PS metrics for a sample function and file in the Latex2rtf project at a single revision.

We used the same approach to compute the PS and UC metrics at the file and function level respectively for 887 revisions of the Apache HTTP version 1.3 project. The revisions of the Apache HTTP project were retrieved from the Subversion repository for the Apache HTTP project. Compared with the Latex2rtf project, we examine many more revisions of the Apache HTTP project. This is due to the Apache HTTP project having many more revisions in its repository, and some Latex2rtf revisions not being compilable. After the PS and UC metrics were generated for the 877 Apache project revisions, we obtained the PS and UC metrics value set for each file and function.

4.3. Bug labeling

Since we will evaluate the accuracy of bug prediction using PS or UC metrics, we need to determine whether a file or a function examined in the experiment actually contains bugs. We label as buggy those project files or functions that actually have at least one bug during the revisions examined. To perform bug labeling, we assume that a file or function has a bug if it has one or more bug-introducing changes in its examined change history. Before we identify bug-introducing changes, we identify bug-fix changes based on the log messages that are supplied with a change. There are two approaches for this step: looking for keywords like "Fixed" or "Bug" in the change log, a technique introduced by Mockus and Votta [18], or looking for references to bug reports like "#42233" as introduced by Fischer et al. [19] and by Cubranic and Murphy [20]. We use the 'looking for keywords' method. If a change log contains 'bug', 'fix', or 'patch', we assume the change is a bug-fix change. To identify bug-introducing changes, we annotate each line of the preceding revision with the most recent revision that changed this line. In this manner we can trace lines backwards through the revision history. We used this method to perform bug labeling for 76 revisions of Latex2rtf project and 887 revisions of the Apache project, identifying 20 buggy files and 132 buggy functions in Latex2rtf, and 16 buggy files and 67 buggy functions in Apache, as shown in Table 1.

Table 1. Analyzed projects.

Project	Period	# of revisions	# of files	# of actually buggy files	# of functions	# of actually buggy functions
Apache HTTP	04/1998 ~ 09/1998	887	46	16	813	67
Latex2rtf	10/2002 ~ 11/2005	76	25	20	524	132

Table 2. Average metric value of PS metrics for a Latex2rtf function and a Latex2rtf file at one revision. (C1: SliceCount, C2: verticesCount, C3: edgesCount, C4: edgesToVerticesRatio, C5: sliceVerticesSum, C6: maxSliceVertices, C7: globalInput, C8: globalOutput, C9: directFanIn, C10: indirectFanIn, C11: directFanOut, C12: indirectFanOut, C13: lackOfCohesion**)**

function or file name	Revision	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13
setPackag()	4/17/2005	6	552	1447	2.62	1120	278	43	2	2	8	2	5	1.94
cfg.c	10/1/2004	40	4794	12742	2.66	1608	402	588	17	189	327	39	205	2.3

4.4. Bug classification

In the experiment, we use PS and UC metrics to perform buggy file or function classification prediction. There are eight bug classification experiments, listed below, exploring all combinations of metric kinds (PS or UC), metric granularity levels (file or function), and project (Latex2rtf or Apache):

Bug file classification prediction using PS metrics at file level for the Latex2rtf project.

Bug function classification prediction using PS metrics at function level for the Latex2rtf project.

Bug file classification prediction using UC metrics at file level for the Latex2rtf project.

Bug function classification prediction using UC metrics at function level for the Latex2rtf project.

Bug file classification prediction using PS metrics at file level for the Apache project.

Bug function classification prediction using PS metrics at function level for the Apache project.

Bug file classification prediction using UC metrics at file level for the Apache project.

Bug function classification prediction using UC metrics at function level for the Apache project.

We use the PS metrics at the file level for the Latex2rtf project as an example for explaining the process we used to perform bug classification prediction. The other bug classification experiments follow the same process.

We use the 10-fold cross-validation method [15] to make the buggy/bug-free classification prediction based on the PS metrics for all 25 Latex2rtf files. In the 10-fold cross-validation method, we randomly divide the 25 Latex2rtf files into 10 folds as shown in Figure 3. We select the first fold as the test set, and the others as the training set. We use a machine learning algorithm to predict the buggy files in the test set. To perform prediction, the machine learning algorithm uses the training set to determine the correlation of the PS metrics for these files with the bug facts obtained in the previous step. Then, it uses the correlation between metrics and bugs to predict which files in the test set will contain bugs. After the bug classification prediction, we compare the prediction results for the test set with the real bug facts obtained from the bug labeling step to determine the prediction accuracy for the files in the test set.

For example, suppose there is a file *cfg.c* in the test set. Based on its PS metrics, and the correction between the PS metrics and bug labeling for the files in the training set, we predict it to be a buggy file. Then, we compare the bug prediction for *cfg.c* with its real bug label as computed in the Bug Labeling step. If they are the same, we call this a correct prediction for *cfg.c*; otherwise, the prediction is incorrect.

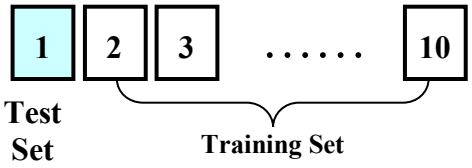

Figure 3. 10-fold cross-validation.

The method iterates by selecting the 2nd, 3rd …10th fold as a test set, and others as training set. After 10 iterations, we obtain the bug classification prediction result of PS metrics for each of the 46 files. The accuracy of the prediction is determined by the formulas in Figure 4.

$$\text{Prediction precision for buggy files} = \frac{\text{Number of files that are predicted to be buggy and are actually buggy}}{\text{Number of files predicted to be buggy}}$$

$$\text{Prediction precision for bug-free files} = \frac{\text{Number of files that are predicted to be bug-free and are actually bug-free}}{\text{Number of files predicted to be bug-free}}$$

$$\text{Prediction recall for buggy files} = \frac{\text{Number of files that are predicted to be buggy and are actually buggy}}{\text{Number of files actually buggy}}$$

$$\text{Prediction recall for bug-free files} = \frac{\text{Number of files that are predicted to be bug-free and are actually bug-free}}{\text{Number of files actually bug-free}}$$

$$\text{Overall prediction accuracy} = \frac{\text{Number of files that are correctly predicted to be buggy or bug-free}}{\text{Number of files}}$$

Figure 4. Prediction precision and recall.

In the 10-fold cross-validation procedure, the machine learning algorithm applied is a Bayesian network classifier [21], which uses graph models to represent interactions between features and classes. Based on the

model, the classifier computes probabilities for each class, and determines the class with the maximum probability. In our experiment, we used the Bayesian network classifier implementation in the Weka data mining software [15].

For the PS metrics at the function level, we used the same procedure to perform bug classification prediction for every function in the Latex2rtf project. That is, we divided the 524 functions in the Latex2rtf project into 10 folds. We took each fold as a test set and the other 9 folds as training sets to predict the bug or bug-free property of each function in the test set using the PS metrics at the function level. After 10 iterations, we obtained the bug classification prediction for each of the 524 functions. The accuracy of the prediction is computed using formulas similar to those in Figure 4, where 'number of files' is replaced by 'number of functions'.

We repeated the bug classification experiment using the same process for the other combinations, i.e. PS metrics for the Apache project, UC metrics for the Apache project, and UC metrics for the Latex2rft project.

4.5. Experimental Results

Figures 5 and 6 show the results of bug classification prediction using PS and UC metrics for the HTTP project. Figures 7 and 8 show the results for the Latex2rtf project. Tables 3 and 4 show the number of correct classifications made using PS metrics and UC metrics, which are used to compute the overall accuracy in Figures 5 through 8. For example, using PS metrics we correctly classified 38 out of 46 files, so its overall accuracy is 82.6% (38/46).

Table 3. Correct classifications of PS and UC at the file level

	# of Files	# of correct classifications by PS	Accuracy of PS (overall)	# of correct classifications by UC	Accuracy of UC (overall)
Apache	46	38	82.6%	37	80.4%
Latex2rtf	25	23	92%	22	88%

Table 4. Correct classifications of PS and UC at the function level

	# of Functions	# of correct classifications by PS	Overall accuracy of PS	# of correct classifications by UC	Overall accuracy of UC
Apache	823	643	78.1%	614	74.6%
Latex2rtf	524	373	71.2%	369	70.4%

Generally, the PS metrics have slightly better performance than the UC metrics: in the Apache experiment, the overall precision of PS metrics at the file level is 2.2% higher (82.6% vs. 80.4%) than the UC metrics at the file level, while the PS metrics at the function level are 3.5% higher (78.1% vs. 74.6%) than the UC metrics. In the Latex2rtf experiment, the overall precision of PS metrics at the file level is 4% higher (92% vs. 88%) than the UC metrics, while the PS metrics at the function level have almost the same performance (71.2% vs. 70.4%) as the UC metrics.

In Figures 5 and 7, the PS metrics outperform UC metrics on almost all comparison items: the precision and recall for buggy or bug-free files. In Figures 6 and 8, the PS metrics have a lower recall rate on buggy functions and a lower precision when predicting bug-free functions, but PS has better overall predictive accuracy than the UC metrics.

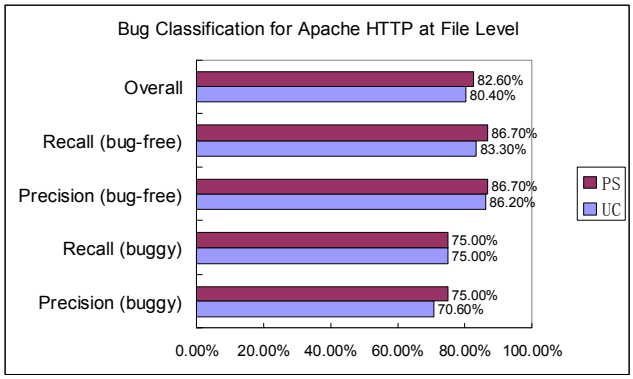

Figure 5. Comparison of PS metrics and UC metrics at file level for the Apache project.

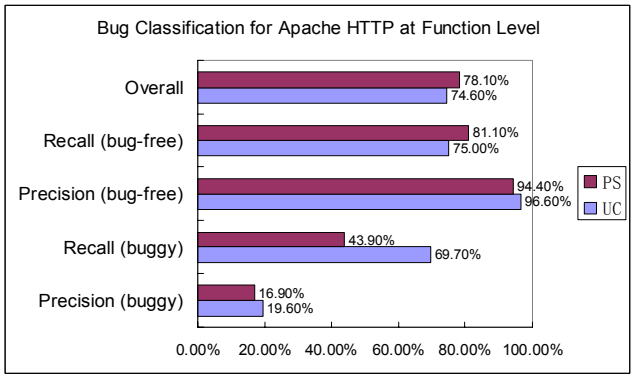

Figure 6. Comparison of PS metrics and UC metrics at function level for the Apache project.

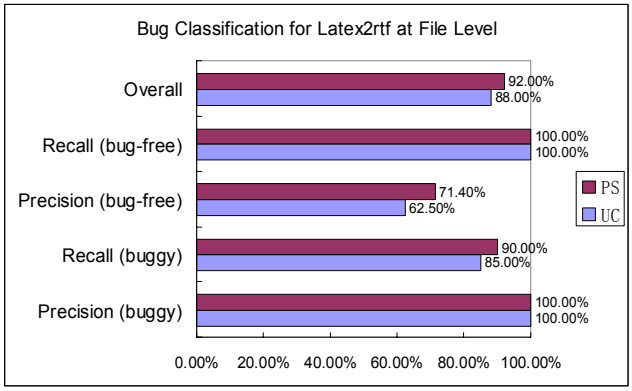

Figure 7. Comparison of PS metrics and UC metrics at file level for the Latex2rtf project.

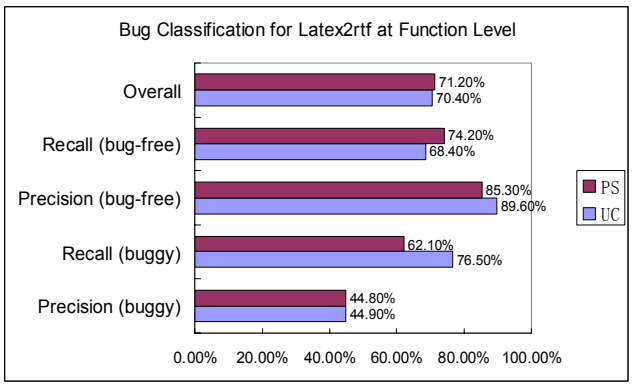

Figure 8. Comparison of PS metrics and UC metrics at function level for the Latex2rtf project.

We used the ChiSquare evaluator in the Weka data mining software [15] to measure the usefulness rank of each PS metric in classification. We list the top 6 PS metrics that are the most useful in the classification in Table 5. Generally, the most useful metrics vary in different projects and granularities, but the metrics *verticesCount* and *edgeCount* are useful metrics in most of the combinations.

We note that file level classification has different usefulness of PS metrics than the function level. For file level classification, *edgesCount* and *directFanOut* are very important metrics in the two projects as shown in Table 5, while they are not in the function level classification. We need to perform further analysis to see if these differences among significant metrics at the file and function level hold true across multiple projects. This is future work.

Table 5. Top 6 of the most useful PS metrics in classification. (a: average, d: accumulative value difference, s: standard deviation)

	File Level	Function Level
HTTP	d-verticesCount a-verticesCount d-edgesCount a-edgesCount d-directFanOut a-globalOutput	a-lackOfCohesion d-lackOfCohesion a-verticesCount a-maxSliceVertices d-verticesCount a-sliceVerticesSum
Latex2rtf	d-edgesCount a-edgesCount d-directFanOut a-directFanOut d-indirectFanOut a-indirectFanOut	d-edgesToVerticesRatio d-sliceCount a-sliceCount s-lackOfCohesion s-verticesCount s-edgesCount

5. Discussion

5.1. Pros and Cons of PS Metrics

Unlike straightforward code metrics based on line counts and statement counts, the PS metrics consider more insightful code properties based on program behaviors, as captured by program slices and obtained from program analysis and points-to analysis. The PS metrics give different weights to each statement based on their significance in the control dependence and flow dependence in the program. For example, a *while* predicate that encloses multiple statements will contribute more than one control dependence edges in PS metrics, while it only contributes one source code line in UC metrics. For function coupling, the finer-grained couplings between function slices, e.g. the couplings between program slices in caller functions and in callee functions are considered, instead of the coupling between whole functions. As a result, the PS metrics summarize properties of the code more precisely. Additionally, the points-to analysis helps the PS metrics recover program complexity and coupling hidden by pointer variables. It makes sense that a function containing many uses or modifications of pointer variables should be viewed as more complex and error-prone than a function without such use.

The major disadvantage of PS metrics is obvious: it is expensive to compute the program slicing information for a project. For example, it took about 20 minutes for CodeSurfer to analyze one revision of the Apache project, which has about 70K lines of source code, on a Pentium 4, 1.7G HZ Linux system with 768 MB memory using the default options of CoderSurfer. This is the reason that we only analyzed a short period of the Apache project in our experiments. Another disadvantage of PS metrics is the program needs to be compiled to compute them, and so

we are unable to compute PS metrics for revisions containing syntax errors.

To address the shortcomings of PS metrics, one possibility is to use a lightweight program analysis implementation to generate the PS metrics. A lightweight program analysis would hopefully achieve a good balance between precision and efficiency, allowing the computation of PS metrics for larger projects. Exploring this possibility remains future work.

5.2. Precision and Recall Rates

We observe in Figure 6 that both PS and UC metrics at the function level have low precision in predicting buggy functions for the Apache project, at 16.9% (PS) and 19.6% (UC) precision respectively. One explanation is that the percentage of buggy functions is low for this project: only 8.2% (67 out of 813) of functions are buggy in the Apache project. As a result, it is difficult to make bug classification predictions due to the rarity of the buggy functions. Even so, the classification prediction precisions from PS and UC metrics are still much higher than that from a random-guess classification. That is, if we randomly classify a function as buggy or bug-free, the classification precision for buggy functions in the HTTP project is about 8.2% (67 out of 813) compared with 16.9% (PS) and 19.6% (UC). In contrast, both PS and UC metrics achieved 100% precision on buggy files and 100% recall on bug-free files for the Latex2rtf project, as shown in Figure 7. This is partially due to the fact that there are only 5 bug-free files in the project, so it is easy to achieve high recall; most files (80%) are buggy, so it is easy to achieve high precision.

5.3. Threats to validity

In our experiment, we only examined a specific revision period for each project. The examined revision periods of the projects or the projects themselves in our experiment may not be representative. Additionally, due to the time and computational expense of computing program slice information over multiple project revisions, our data set only includes two projects. In general, the larger the dataset, the more accurate the evaluation.

In addition, the revision set in experiment is not complete for both HTTP project and Latex2rtf project, since some revisions are not compilable.

6. Conclusions and future work

In this paper, we described program slicing metrics, a set of code metrics for C programs using program slicing techniques. Program slicing metrics measure the size, complexity, coupling, and cohesion properties of programs based on program slices, vertices in slices, and dependence edges in slices.

To evaluate the program slicing metrics, we use them and Understand for C++ metrics respectively to classify buggy/bug-free source code files or functions for 887 revisions of the Apache HTTP project and 76 revisions of the Latex2rtf project, and compare their results. The outcomes show that the PS metrics have a slightly better performance than UC metrics in classifying buggy/bug-free source code: PS metrics have an overall 82.6% and 92% accuracy at file level for the Apache HTTP project and the Latex2rtf project respectively, and UC metrics have an overall 80.4% precision and 88% precision at the file level for the Apache HTTP project and the Latex2rtf project respectively. The results show that the PS metrics have at least the same power as UC metrics in performing bug classification.

A major problem of PS metrics is that it is expensive to generate the program slicing information for a large project. To address this problem in our future work, we will choose and apply light-weight program analysis in computing PS metrics. For example, we can choose a less expensive pointer analysis method when performing program analysis for a project.

We also need to make more experiments on other projects to further validate PS metrics. Especially when a light-weight program analysis is used, the experiment can be performed on a longer version history of larger projects.

7. References

[1] T. J. McCabe and A. H. Watson, "Software Complexity," Crosstalk, vol. 7, no. 12, pp. 5-9, 1994.

[2] M. Weiser, "Program Slicing," IEEE Transactions on Software Engineering, vol. 10, no. 4, pp. 352-357, 1984.

[3] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," ACM Transactions on Programming Languages and Systems, vol. 12, no. 1, pp. 26-60, 1990.

[4] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The Program Dependence Graph and its Use in Optimization," ACM Transactions on Programming Languages and Systems, vol. 9, no. 3, pp. 319-349, 1987.

[5] S. Toolworks, "Maintenance, Understanding, Metrics and Documentation Tools for Ada, C, C++, Java, and FORTRAN," 2006, http://www.scitools.com/.

[6] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction," Transactions on Software Engineering, vol. 31, no., pp. 897-910, 2005.

[7] T. M. Khoshgoftaar and E. B. Allen, "Predicting the Order of Fault-prone Modules in Legacy Software," In Proceedings of the Ninth International Symposium on Software Reliability Engineering, Paderborn, Germany, 1998, pp. 344-353.

[8] T. M. Khoshgoftaar and E. B. Allen, "Ordering Fault-Prone Software Modules," Software Quality Journal, vol. 11, no. 1, pp. 19-37, 2003.

[9] J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey, "Facilitating Software Evolution with Kenyon," In Proceedings of the 2005 European Software Engineering Conference and 2005 Foundations of Software Engineering (ESEC/FSE 2005), Lisbon, Portugal, 2005, pp. 177-186.
[10] L. Ott and J. Thuss, "Slice Based Metrics for Estimating Cohesion," In Proceedings of the First International Software Metrics Symposium, 1993, pp. 71-81.
[11] J. M. Bieman and L. M. Ott, "Measuring Functional Cohesion," IEEE Transactions on Software Engineering, vol. 20, no. 8, pp. 644-657, 1994.
[12] B. Li, "A Hierarchical Slice-Based Framework for Object-Oriented Coupling Measurement", TUCS Technical Report No.415, Turku Center for Computer Science, Abo Akademi University, 2001.
[13] T. M. Meyers and D. Binkley, "Slice-Based Cohesion Metrics and Software Intervention," In Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04), Delft, The Netherlands, 2004, pp. 256-265.
[14] J. Sliwerski, T. Zimmermann, and A. Zeller, "When Do Changes Induce Fixes?," In Proceedings of Int'l Workshop on Mining Software Repositories (MSR 2005), Saint Louis, Missouri, 2005, pp. 24-28.
[15] I. H. Witten and E. Frank, Data Mining: Practical machine learning tools and techniques (Second Edition), Morgan Kaufmann, 2005.
[16] Latex2rtf, "LaTeX to RTF converter Project Home Page," 2006, http://sourceforge.net/projects/latex2rtf/.
[17] GrammaTech, "GrammaTech CodeSurfer Home Page," 2006, http://www.grammatech.com/products/codesurfer/index.html.
[18] A. Mockus and L. G. Votta, "Identifying Reasons for Software Changes using Historic Databases," In Proceedings of International Conference on Software Maintenance (ICSM 2000), San Jose, California, 2000, pp. 120-130.
[19] M. Fischer, M. Pinzger, and H. Gall, "Populating a Release History Database from Version Control and Bug Tracking Systems," In Proceedings of 2003 Int'l Conference on Software Maintenance (ICSM'03), 2003, pp. 23-32.
[20] D. Cubranic and G. C. Murphy, "Hipikat: Recommending Pertinent Software Development Artifacts," In Proceedings of the 25th International Conference on Software Engineering, Portland, Oregon, 2003, pp. 408 - 418.
[21] E. Alpaydin, Introduction to Machine Learning, MIT Press, 2004.