Statement-Level Cohesion Metrics and their Visualization

Jens Krinke
FernUniversität in Hagen
Hagen, Germany

Abstract

Slice-based metrics for cohesion have been defined and examined for years. However, if a module with low cohesion has been identified, the metrics cannot help the maintainer to restructure the module to improve the cohesion. This work presents statement-level cohesion metrics based on slices and chops. When visualized, the statement-level cohesion metrics can show which parts of a module have a low cohesion and thus help the maintainer to identify the parts that should be restructured.

1. Introduction

An important principle for software development is that modules should have high cohesion (inside modules) and low coupling (to other modules). Cohesion is in most parts a subjective measure, however, functional cohesion (the most desirable of cohesion categories) can be measured based on the dependences that can be identified in the source code. Slice-based cohesion metrics exist for years now and have been researched extensively [1, 5, 8, 18–22, 25]. These metrics only measure the cohesion of a complete module (i.e. procedure, function, method, unit, class, ...), however, if a module with low cohesion has been identified, the metrics cannot help the user to restructure the module to improve its cohesion.

Although the extraction of code from procedures with low cohesion can be automated [9, 10, 12–15], there is still need to identify the code's regions that can be extracted to improve the procedure's cohesion. This work will present a new approach for statement-level cohesion metrics based on slices and chops. The metrics are based on the previous work on functional cohesion and the slice profiles used therein. The approach computes a cohesiveness value for every statement in a program and visualizes it by coloring the source code. This visualization intuitively presents the regions inside a procedure that are responsible for the low overall cohesion of the procedure to a maintainer. The approach is even able to suggest independent regions in the source code with low cohesion as candidates for extraction.

In the next section, the background on dependence graphs, slicing, and cohesion metrics will be recapitulated. Section 3 presents statement-level cohesion metrics and how they are visualized. Section 4 discusses related work and the last section concludes with the implementation's current status.

2. Slices and Metrics

Ottenstein and Ottenstein [23] were the first to suggest the use of program dependence graphs to compute Weiser's slices. Program dependence graphs mainly consist of nodes representing the program's statements along with control and data dependence edges:

- *Control dependence* between two statement nodes exists if one statement controls the execution of the other.
- *Data dependence* between two statement nodes exists if a variable's definition at one statement might reach the same variable's usage at another statement.

Slicing without procedures is trivial: Just find reachable nodes in the PDG [3]. The underlying assumption is that all paths are *realizable*. This means that a possible execution of the program exists for any path that executes the statements in the same order.

The (*backward*) *slice* $S(n)$ of a PDG at node n consists of all nodes on which n (transitively) depends:

$$S(n) = \{m \mid m \rightarrow^{\star} n\}$$

The node n is called the *slicing criterion*.

In presence of procedures, slicing is no longer trivial in PDGs. If the calling context is ignored, the analysis is *context-insensitive*, as a called procedure may return to a call site different to the site it has been called from. However, if the calling context is obeyed, the results will be much more precise. Paths along transitive dependences are now considered realizable only if they obey the calling context. Thus, slicing is *context-sensitive* if only realizable paths are traversed. Context-sensitive slicing is solvable efficiently—one has to generate summary edges at call

sites [6]: Summary edges represent the transitive dependences of called procedures at call sites.

Weiser proposed several metrics based on slices in his thesis [25], called *tightness*, *overlap*, *coverage*, *parallelism*, and *clustering*. Ott, Thuss, and Bieman [1, 20, 21] formalized most of them and introduced various others. Their studies showed that overlap, coverage, and tightness were the most useful ones. Their formalization uses the following sets based on a module M which can be a procedure or method.

- V_M is the set of variables used by M (occur in M).
- $V_O \subseteq V_M$ is the set of output variables, i.e. variables that are used after the module M is left. For example, these include the return value or global variables.
- SL_x is the slice for a variable $x \in V_O$.
- $SL_{int} = \bigcap_{x \in V_O} SL_x$ is the intersection of all slices for the output variables of M.

SL_x is the backward slice for variable x at the end of the module. When PDGs are used for computation, this is computed by a backward slice from the node that represents the variable x before it is passed back to the actual parameter at the call site.

Based on the above defined sets, the following metrics are defined:

Coverage is a comparison of the length of the slices to the length of the module:

$$Coverage(M) = \frac{1}{|V_O|} \sum_{x \in V_O} \frac{|SL_x|}{length(M)} \qquad (1)$$

Overlap is the average ratio of the number of the statements in the intersection of all slices to the size of each slice:

$$Overlap(M) = \frac{1}{|V_O|} \sum_{x \in V_O} \frac{|SL_{int}|}{|SL_x|} \qquad (2)$$

Tightness is the number of statements included in every slice compared to the number of statements in the module:

$$Tightness(M) = \frac{|SL_{int}|}{length(M)} \qquad (3)$$

For all metrics, higher values represent better cohesion. Ott, Thuss, and Bieman have used *Slice Profiles* to explain the computation of the metrics. A slice profile shows a module's source code together with a column for each slice SL_x that marks the statements included in that slice. Figure 1 shows a running example that computes a sum, a product, and Fibonacci numbers. Its slice profile is shown in the first three columns that show the slice SL_x for the three output variables `s`, `p`, and `f` ($V_O = \{\texttt{s}, \texttt{p}, \texttt{f}\}$). The computed metrics' values for the example are: *Coverage* $= \frac{7}{13}$, *Overlap* $= \frac{3}{7}$, and *Tightness* $= \frac{3}{13}$. It seems that the procedure's cohesiveness is not very good because the numbers are low. The metrics alone give no clue why this is the case.

s	p	f		
			1	`void compute(int a, int b,`
			2	`  int* s, int* p,`
			3	`  int[] f) {`
			4	`  int i;`
●	●	●	5	`  if (a > b) {`
●	●	●	6	`  int t = a;`
●	●		7	`  a = b;`
●	●	●	8	`  b = t;`
			9	`  };`
●			10	`  *s = 0;`
	●		11	`  *p = 1;`
●	●		12	`  for (i = a; i<=b; ++i) {`
●			13	`  *s += i;`
	●		14	`  *p *= i;`
			15	`  };`
		●	16	`  f[0] = 1;`
		●	17	`  f[1] = 1;`
		●	18	`  for (i = 2; i<=b; ++i) {`
		●	19	`  f[i] = f[i-1] + f[i-2];`
			20	`  };`
			21	`};`

Figure 1. A slice profile

Longworth [18] already noted some inconsistencies when applying the metrics. To improve the usefulness, the meaning of a slice SL_x has been changed and the results have been studied. Ott and Thuss [22] have used *metric* slices, which are the union of forward and backward slices for the output variables. In the example, there is no difference between normal and metric slices. Later, Bieman and Ott have refined metric slices further to *data* slices, which are based on *tokens*, i.e. variables and constant definitions and references. Harman et al. also refined the slices to be more fine-grained, for example by using the number of distinct variables in an expression [5].

All metrics are targeted to measure the cohesion of a module (procedure, method, class, unit, etc.). However, all previous work only has shown (small) examples and none showed that the computed values really measure the cohesion. Moreover, these metrics cannot help the user to restructure the module to improve its cohesion. Thus, the next section will present statement-level cohesion that can guide the restructuring of modules with low cohesion.

3. Statement-Level Cohesion

After a 'suspicious' value of a computed cohesion metric has been identified for a module, the metrics cannot help a software maintainer to improve the module's structure. However, the slice profiles can be used to explain the metrics. A slice profile like the one in Figure 1 can immediately give an overall impression of every statement's cohesiveness: the upper part seems to be very cohesive while the lower part seems to have low cohesion. However, such a visualization is not suitable for large modules with many slices. Therefore, this section presents a new approach which uses the idea of slice profiles to define Statement-Level Cohesion. Later on, this will be used for visualization and for identification of code regions with low cohesion which can be extracted to improve the module's cohesion.

We define the cohesiveness of a statement $s \in M$ as the number of slices in which it is included in comparison to the total number of slices:

$$(Simple)\,Cohesiveness_{SL}(s) = \frac{|\{x \mid s \in SL_x\}|}{|V_O|} \quad (4)$$

The higher the computed value for a statement, the more cohesive is the statement to other statements of the module. At the statement level, there is no clear distinction between cohesion and coupling: a cohesive region consists of coupled statements. Thus, from now on we will only use the term cohesion (or cohesiveness). Figure 2 shows the computed cohesiveness for each statement of the example. This clearly shows that the upper part which swaps `a` and `b` is very cohesive while the lower part is not (two independent computations are performed there).

The first definition of cohesiveness in Equation 4 does not consider the slices' size. However, this can produce misleading results as Figure 3 shows. The procedure in the figure has the metric values *Coverage* = $\frac{1}{2}$, *Overlap* = 0, and *Tightness* = 0, which suggests that the procedure has low cohesion. Here, all statements have a cohesiveness of $\frac{1}{2}$ (shown in the first column) but most of the procedure is indeed cohesive. The values arise because a single statement that is unrelated to the rest of the procedure exists. However, none of the metrics is able to identify that this is the case let alone which statement is responsible.

The improved metric includes the size of the slices:

$$Cohesiveness_{SL}(s) = \frac{\sum_{x|s\in SL_x} |SL_x|}{\sum_{x\in V_O} |SL_x|} \quad (5)$$

This is basically the ratio of the size of all slices to which the statement belongs and the size of all slices in the module. The last column in Figure 3 shows the improved metric: it is now obvious that only the line '`*p = 1`' is not cohesive to the rest of the procedure. The metrics' values for

$\frac{x}{3}$		
	1	`void compute(int a, int b,`
	2	`int* s, int* p,`
	3	`int[] f) {`
	4	`  int i;`
3	5	`  if (a > b) {`
3	6	`    int t = a;`
2	7	`    a = b;`
3	8	`    b = t;`
	9	`  };`
1	10	`  *s = 0;`
1	11	`  *p = 1;`
2	12	`  for (i = a; i<=b; ++i) {`
1	13	`    *s += i;`
1	14	`    *p *= i;`
	15	`  };`
1	16	`  f[0] = 1;`
1	17	`  f[1] = 1;`
1	18	`  for (i = 2; i<=b; ++i) {`
1	19	`    f[i] = f[i-1] + f[i-2];`
	20	`  };`
	21	`};`

Figure 2. Cohesiveness of the statements

$\frac{x}{2}$			$\frac{x}{8}$
	1	`void compute(int a, int b,`	
	2	`int* s, int* p,`	
	3	`int[] f) {`	
	4	`  int i;`	
1	5	`  if (a > b) {`	7
1	6	`    int t = a;`	7
1	7	`    a = b;`	7
1	8	`    b = t;`	7
	9	`  };`	
1	10	`  *s = 0;`	7
1	11	`  *p = 1;`	1
1	12	`  for (i = a; i<=b; ++i) {`	7
1	13	`    *s += i;`	7
	14	`  };`	
	15	`};`	

Figure 3. (Un)weighted Cohesiveness

a	b	f	s	p	f	a	b	s	a	f	p	a	b	f	p	a	b	f	f	$\frac{x}{9}$	$\frac{x}{35}$	
																						`void compute(int a, int b,`
																						`int* s, int* p,`
																						`int[] f) {`
																						`int i;`
|	|		|	|	|		|	|	|		|	|	|		|	|		6	34	`if (a > b) {`
|	|		|	|	|		|	|	|		|	|	|		|	|		6	34	`int t = a;`
|	|		|	|	|				|	|				|				4	24	`a = b;`
|	|		|	|	|		|	|	|		|	|	|		|	|		6	34	`b = t;`
																						`};`
		|																			0	0	`*s = 0;`
			|																		0	0	`*p = 1;`
|	|		|	|	|		|	|											4	23	`for (i = a; i<=b; ++i) {`
|	|		|				|	|												2	11	`*s += i;`
|	|			|						|	|									2	12	`*p *= i;`
																						`};`
					|																0	0	`f[0] = 1;`
					|																0	0	`f[1] = 1;`
|	|				|									|	|					2	11	`for (i = 2; i<=b; ++i) {`
|	|	|			|									|	|	|			3	12	`f[i] = f[i-1] + f[i-2];`
																						`};`
																						`};`

Figure 4. A chop profile

the example in Figure 2 are the same as before, because the three slices have the same size of seven elements, but this is just incidental.

Note that metric slices have been used in the above definitions. The specific kind of the used slice is changeable. To compute the slices SL_x one can use traditional slices, metric slices, data slices, etc. Another kind of slice seems to be valuable in this context: Jackson and Rollins [7] introduced *Chopping*, which reveals the statements involved in a transitive dependence from one specific statement (the source criterion) to another (the target criterion). For the purpose of cohesion metrics, the slices are now replaced by chops between the input and the output variables. Let V_I be the set of input variables, i.e. procedure arguments and global variables ($V_I \subseteq V_M$) and let $CH_{x,y}$ be the chop from $x \in V_I$ at the start of the module to $y \in V_O$ at the end of the module.

The cohesiveness of a statement $s \in M$ is then the number of chops in which it is included in comparison to the number of all chops:

$$(Simple)\,Cohesiveness_{CH}(s) = \frac{|\{(y,x) \mid s \in CH_{y,x}\}|}{|V_I||V_O|} \quad (6)$$

Again, inclusion of the chops' sizes leads to a better and more expressive metric:

$$Cohesiveness_{CH}(s) = \frac{\sum_{(y,x)|s \in CH_{y,x}} |CH_{y,x}|}{\sum_{(y,x)|y \in V_I, x \in V_O} |CH_{y,x}|} \quad (7)$$

Figure 4 shows a chop profile for the example. Remember that the output variables are $V_O = \{\mathtt{s}, \mathtt{p}, \mathtt{f}\}$. We require that the underlying analysis can clearly identify the input variables [4, 11, 17], thus $V_I = \{\mathtt{a}, \mathtt{b}, \mathtt{f}\}$. Note that the array is an input variable because it is only partially modified (only the first b array elements are overwritten and the rest could be used after the procedure returns). Because a chop inside a procedure is just the intersection of a forward and a backward slice, the figure shows the forward slice profile for the input variables in the first three columns and the backward slice profile in the next three columns. Nine chops have to be computed and the next nine columns show the chop profile. The columns $\frac{x}{9}$ and $\frac{x}{35}$ show both versions of $Cohesiveness_{CH}$, which only slightly differ.

The chop-based cohesiveness has different values to the slice-based cohesiveness and a very interesting property: The statements which assign a constant to the variables `s`, `p`, and `f` all have a zero cohesiveness. They all have neither a data dependence to an input variable nor a control dependence, and thus, they cannot be part of any chop.

Again, the specific kind of chop can be changed (traditional or token-based), however, a metric chop does not make sense.

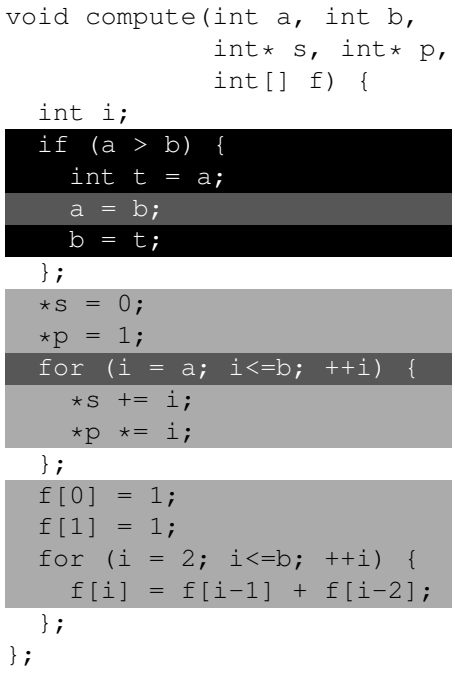

Figure 5. Visualization of the Cohesiveness

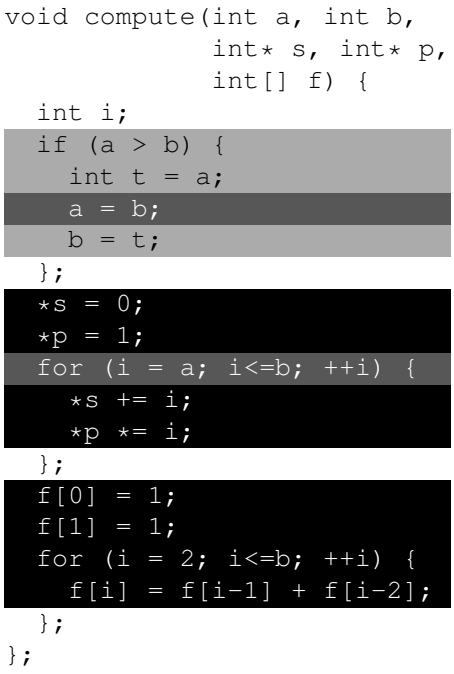

Figure 6. Visualization in negative mode

Visualization

A statement-level metric is of no use without an appropriate visualization, because of the amount of data. We have implemented such a visualization inside our slicing system. This visualization basically colorizes the source code based on the computed cohesiveness values for all statements. Figure 5 shows the visualization for the example.

The visualization can be used in *positive* and in *negative* mode: In positive mode, the source code with higher cohesiveness is darker and in negative mode, it is lighter. We have found the negative mode to be more useful as it highlights the parts of the source code with low cohesiveness. Figure 6 shows the same visualization in negative mode and it is obvious that the procedure's larger part has a low cohesion and should be restructured.

What cannot be shown here is a green/red visualization: This visualization uses a color scale that moves from green to red instead of light to dark. With this visualization, statements with high cohesion are colored green and statement with low cohesion are colored red.

Although only the visualization in the source code is shown, this type of visualization can be used in a Seesoft [2] visualization, where each line of code is mapped into a thin colored row and each file is mapped into a small column.

In the following, the visualization will not be used in the figures because of clearer presentation. Instead, the numerical values of the metrics will be shown.

Suggestions for Restructuring

The visualization alone is able to show areas inside a procedure which are not very cohesive. However, if such areas exist it is desirable to suggest to the maintainer how they can be removed to improve the procedure's cohesion. The following shows how this can be achieved.

The first step is to remove all interprocedural nodes (i.e. actual-in, -out, and entry nodes) and edges (i.e. parameter-in, -out, and call edges) from the PDG because only areas inside a procedure are to be removed or restructured. The next step is to remove all nodes (and incident edges) from the PDG that correspond to statements with a high cohesiveness. This requires that the maintainer specifies a limit c for cohesiveness. After all nodes corresponding to statements s with $Cohesiveness(s) > c$ have been removed, the remaining graph consists of a set of isolated regions, i.e. components that are not connected by a control or data dependence edge. The isolated components are regions with low cohesion and each of the components is a candidate for extraction into a new module. The extraction not only improves the restructured module's cohesion but also ensures a high cohesion of the new modules.

When applied to the example in Figure 2, the removal of all interprocedural nodes and edges results in the PDG shown in Figure 7, where the gray nodes have a cohesiveness higher than $c = \frac{1}{3}$. The second step, the removal of all nodes (resp. statements) with a high cohesiveness (the gray

```
1  void compute(int a, int b,
2               int* s, int* p,
3               int[] f) {
4    int i;
5    if (a > b) {
6      int t = a;
7      a = b;
8      b = t;
9    };
10   *s = 0;
11   *p = 1;
12   for (i = a; i<=b; ++i) {
13     *s += i;
14     *p *= i;
15   };
16   f[0] = 1;
17   f[1] = 1;
18   for (i = 2; i<=b; ++i) {
19     f[i] = f[i-1] + f[i-2];
20   };
21 };
```

(a) removal of statements

```
*s = 0;
  *s += i;
```

(b) component 1

```
*p = 1;
  *p *= i;
```

(c) component 2

```
f[0] = 1;
f[1] = 1;
for (i = 2; i<=b; ++i) {
  f[i] = f[i-1] + f[i-2];
```

(d) component 3

Figure 8. Components after removal of highly cohesive statements

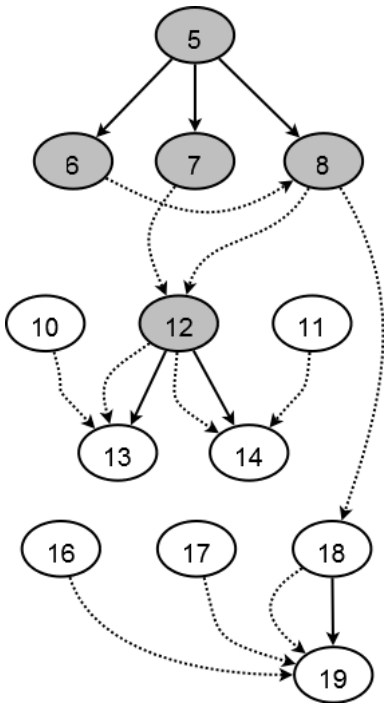

Figure 7. PDG for the example

nodes) together with the incident edges results in three isolated regions that are not connected by a dependence edge: (1) nodes 10 and 13, (2) nodes 11 and 14, and (3) nodes 16 – 19. When visualized in the source code, only the highlighted areas in Figure 8 are not removed. The three connected components can be shown independently to the maintainer (b, c, and d); each of the component reveals a more or less independent computation. The maintainer can then decide that only component (d) is to be extracted. Figure 9 shows the example after restructuring; component (d) has been extracted to the new procedure `fibonacci`, where all statements are now fully cohesive. The statements in the now smaller procedure `compute` have at least a cohesiveness of $\frac{1}{2}$. That `compute`'s cohesion has improved is also visible in the new metrics' better values: *Coverage* $= \frac{7}{9}$, *Overlap* $= \frac{5}{7}$, and *Tightness* $= \frac{5}{9}$. Because `fibonacci` has one single output variable, its metrics' values are all 1. This restructuring has replaced a procedure with low cohesion by two procedures with high cohesion.

The maintainer could also extract components (b) and (c), which would require the duplication and extraction of the for-statement in line 12. This restructuring would result in four procedures, each with a single task. All four procedures would have fully cohesive statements and full coverage, overlap, and tightness. However, he also might not want to do this, because components (b) and (c) have a low functional cohesion but a high logical cohesion.

$\frac{x}{2}$	
	1 `void compute(int a, int b,`
	2 `int* s, int* p) {`
	3 `int i;`
2	4 `if (a > b) {`
2	5 `int t = a;`
2	6 `a = b;`
2	7 `b = t;`
	8 `};`
1	9 `*s = 0;`
1	10 `*p = 1;`
2	11 `for (i = a; i<=b; ++i) {`
1	12 `*s += i;`
1	13 `*p *= i;`
	14 `};`
	15 `};`

$\frac{x}{1}$	
	1 `void fibonacci(int b, int[] f) {`
	2 `int i;`
1	3 `f[0] = 1;`
1	4 `f[1] = 1;`
1	5 `for (i = 2; i<=b; ++i) {`
1	6 `f[i] = f[i-1] + f[i-2];`
	7 `};`
	8 `};`

Figure 9. Cohesiveness after restructuring

Examples

We now present three examples taken from Bieman and Ott's article [1] to illustrate the slice-based metrics and compare metric slices to normal backward slices. Figures 10, 11, and 12 show two modules `decode` and `lookup` which are 'merged' to a third one. To the left of the source code, the profile based on metric slices is shown together with the cohesiveness of each statement as computed by the presented approach. To the right of the source code the same data is given based on normal backward slices. Each of the figures also give the values for coverage, overlap, and tightness (based on the metric slices). It can be seen that the three metrics have high values for the first two examples. In the merged version, coverage is still very high, but overlap and tightness are not. The low value for tightness indicates the presence of a low cohesive area.

When the statements' cohesiveness is examined, there are differences between metric and backward slices. The backward slices in Figure 10 reveal that the computations for `v` and `s` are more or less independent (which is true; however, they are *logically* cohesive). In contrast, the metric slices result in different, but higher cohesiveness values for the assignments to `v` and `s`. In Figure 11, the case is a little bit different: the backward slices result in lower values for two statements in comparison to the values resulting from metric slices.

A similar behavior can be observed in the merged procedure shown in Figure 12. Because of the merge, the tightness of this procedure has dropped. From the computed cohesiveness values for the statements, it is immediately visible which statements are responsible for the drop. For the backward slices, the set of statements with low cohesion is even larger than for metric slices.

Moreover, while the overall metrics for coverage, overlap, and tightness are all still above $\frac{1}{2}$, the statement level cohesion metrics reveal that the statements merged from `decode` are a region of low cohesion.

4. Related Work

Weiser proposed several metrics based on slices in his thesis [25] which he has implemented and evaluated in a prototype. Longworth [18] investigated the slices and found that several of them are related to cohesion. Ott and Thuss [21] studied this relationship based on the processing element flow graph. Though this is related to the program dependence graph, they use Weiser's slicing. Later, the same authors resolved some problems identified by Longworth with the introduction of metric slices [22]. Further improvements have been introduced by Bieman and Ott [1] by using token-based slices instead of statement-based slices. Harman et al. also refined the slices to be more fine-grained, for example by using the number of distinct variables in an expression [5]. The token-based metrics have later been extended to object-oriented languages [20].

The first empirical study by Karstu [8] was limited because of a limited slicer. The only large scale empirical study has been done by Meyers and Binkley [19].

After low cohesive areas have been identified by the presented approach, the extraction or restructuring of the areas can be automated with approaches also based on slicing. Lakhotia and Deprez [12, 13] specifically target restructuring of procedures with low cohesion. Similar automated approaches to extract code regions from procedures have been researched by Lanubile [14, 15] and Komondoor and Horwitz [9, 10].

Pan et al. [24] introduced 13 simple metrics based on program slicing and examined if the metrics can predict that a file or procedure contains bugs. The presented metrics have a slightly higher precision and recall in comparison to traditional metrics.

Closely related to cohesion metrics are metrics that measure coupling. Li et al. [16] examined some slice-based conditions that are necessary for coupling as an initial step for slice-based coupling metrics.

metric				*backward*		
v	s	$\frac{x}{8}$		v	s	$\frac{x}{6}$
			`void decode(int* v, int* s) {`			
5	3	8	`  if (v < 5000) {`	3	3	6
5		5	`  *v = *v * 8 % 10;`	3		3
5	3	8	`  *s = 1;`		3	3
			`  } else {`			
5		5	`  *v = *v % 10;`	3		3
5	3	8	`  *s = 0;`		3	3
			`  }`			
			`};`			

Coverage $= \frac{4}{5}$, *Overlap* $= \frac{4}{5}$, *Tightness* $= \frac{3}{5}$

Figure 10. Procedure Decode

metric					*backward*			
s	p	a	$\frac{x}{22}$		s	p	a	$\frac{x}{20}$
				`void lookup(table A[], int size, keytype key,`				
				`  int* s, int* p, char** a) {`				
8	7	7	22	`  int i = 1;`	6	7	7	20
8	7	7	22	`  *s = 0;`	6	7	7	20
8	7	7	22	`  while ((*s == 0) && (i < size)) {`	6	7	7	20
8	7	7	22	`  if (A[i].name == key) {`	6	7	7	20
8	7	7	22	`  *s = 1;`	6	7	7	20
8	7		15	`  *p = A[i].value;`		7		7
8		7	15	`  *a = A[i].address;`			7	7
				`  } else {`				
8	7	7	22	`  ++i;`	6	7	7	20
				`  }`				
				`  }`				
				`};`				

Coverage $= \frac{11}{12}$, *Overlap* $= \frac{23}{24}$, *Tightness* $= \frac{7}{8}$

Figure 11. Procedure Lookup

5. Conclusions and Further Work

We have presented a technique to measure the cohesiveness on the level of statements based on previous work on slice profiles and cohesion metrics that are computed by slicing. Unlike the previous cohesion metrics that only measure the cohesion of a module (procedure), the presented approach is able to identify areas with low cohesion inside procedures. This enables the maintainer to identify the procedure's regions that should be extracted to improve the procedure's cohesion. The presented approach supports this task by visualizing the cohesiveness by coloring the source code and by suggesting connected components with low cohesion. If the suggested components are extracted, the module cohesion improves and the new modules made from the extracted components have a high cohesion. It is important to note that although we have only focused on procedures as modules, it is straight-forward to apply the same approach to larger units, for example classes or packages.

The approach presented here has been partially implemented inside our slicing system. Currently, the slice-based and the chop-based cohesiveness can be computed and visualized: The system generates the PDG from the source code of a program to be analyzed. After the cohesiveness is computed for every statement, the source code is presented with the source code colored according to the statements' cohesiveness. The implementation of the technique for the suggestion of connected components with low cohesion is underway. As soon as the implementation is complete, the system will enable software maintainers to rapidly identify code areas that should be extracted if the goal is to improve the procedures' cohesion. This will not only enable

metric					*backward*			
s	p	a	$\frac{x}{30}$		s	p	a	$\frac{x}{27}$
				`void lookup2(table A[], int size, keytype key,`				
				`  int* s, int* p, char** a) {`				
11	12	7	30	`  int i = 1;`	10	10	7	27
11	12	7	30	`  *s = 0;`	10	10	7	27
11	12	7	30	`  while ((*s == 0) && (i < size)) {`	10	10	7	27
11	12	7	30	`  if (A[i].name == key) {`	10	10	7	27
11	12	7	30	`  *s = 1;`	10	10	7	27
11	12		23	`  *p = A[i].value;`	10	10		20
11		7	18	`  *a = A[i].address;`			7	7
				`  } else {`				
11	12	7	30	`  ++i;`	10	10	7	27
				`  }`				
				`  }`				
11	12		23	`  if (*p < 5000) {`	10	10		20
	12		12	`  *p = *p * 8 % 10;`		10		10
11	12		23	`  *s = 1;`	10			10
				`  } else {`				
	12		12	`  *p = *p % 10;`		10		10
11	12		23	`  *s = 0;`	10			10
				`  }`				
				`};`				

$Coverage = \frac{10}{13}$, $Overlap = \frac{293}{462}$, $Tightness = \frac{6}{13}$

Figure 12. Procedure Lookup2

the maintainer to constantly monitor the cohesion of procedures during a software's evolution but also allows for fast countermeasures if cohesion degrades.

References

[1] J. M. Bieman and L. M. Ott. Measuring functional cohesion. *IEEE Trans. Softw. Eng.*, 20(8):644–657, Aug. 1994.

[2] S. Eick, J. Steffen, and E. S. Jr. Seesoft - a tool for visualizing line oriented software statistics. *IEEE Trans. Softw. Eng.*, 18(11):957–968, 1992.

[3] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Prog. Lang. Syst.*, 9(3):319–349, July 1987.

[4] I. Forgcs and T. Gyimthy. An efficient interprocedural slicing method for large programs. In *Proceedings of SEKE'97, the 9th International Conference on Software Engineering & Knowledge Engineering*, pages 279–287, 1997.

[5] M. Harman, S. Danicic, B. Sivagurunathan, B. Jones, and Y. Sivagurunathan. Cohesion metrics. In *8th International Quality Week*, May 1995.

[6] S. B. Horwitz, T. W. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Prog. Lang. Syst.*, 12(1):26–60, Jan. 1990.

[7] D. Jackson and E. J. Rollins. A new model of program dependences for reverse engineering. In *Proceedings of the second ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 2–10, 1994.

[8] S. Karstu. An examination of the behavior of slice based cohesion measures. Master's thesis, Department of Computer Science, Michigan Technological University, 1994.

[9] R. Komondoor and S. Horwitz. Semantics-preserving procedure extraction. In *Proc. of 27th ACM Symp. on Principles of Programming Languages (POPL)*, 2000.

[10] R. Komondoor and S. Horwitz. Effective, automatic procedure extraction. In *Proc. of 11th Int. Workshop on Program Comprehension (IWPC)*, 2003.

[11] J. Krinke. *Advanced Slicing of Sequential and Concurrent Programs*. PhD thesis, Universität Passau, Apr. 2003.

[12] A. Lakhotia and J.-C. Deprez. Restructuring programs by tucking statements into functions. *Information and Software Technology*, 40(11–12):677–690, 1998.

[13] A. Lakhotia and J.-C. Deprez. Restructuring functions with low cohesion. In *Working Conference on Reverse Engineering*, pages 36–46, 1999.

[14] F. Lanubile and G. Visaggio. Function recovery based on program slicing. In *Proceedings of the International Conference on Software Maintenance*, pages 396–405, 1993.

[15] F. Lanubile and G. Visaggio. Extracting reusable functions by flow graph-based program slicing. *IEEE Trans. Softw. Eng.*, 23(4):246–259, Apr. 1997.

[16] B. Li, Y. Zhou, J. Mo, and Y. Wang. Analyzing the conditions of coupling existence based on program slicing and some abstract information-flow. In *Proceedings of the 6th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2005)*, pages 96–101, 2005.

[17] P. Livadas and S. Croll. A new algorithm for the calculation of transitive dependences. *Journal of Software Maintenance*, 6:100–127, 1994.

[18] H. D. Longworth. Slice based program metrics. Master's thesis, Michigan Technological University, 1985.

[19] T. Meyers and D. W. Binkley. Slice-based cohesion metrics and software intervention. In *11th IEEE Working Conference on Reverse Engineering (WCRE 2004)*, pages 256–266, Los Alamitos, California, USA, nov 2004. IEEE Computer Society Press.

[20] L. M. Ott and J. M. Bieman. Program slices as an abstraction for cohesion measurement. *Information and Software Technology*, 40(11-12):691–700, 1998.

[21] L. M. Ott and J. J. Thuss. The relationship between slices and module cohesion. In *Proceedings of the* 11^{th} *ACM conference on Software Engineering*, pages 198–204, 1989.

[22] L. M. Ott and J. J. Thuss. Slice based metrics for estimating cohesion. In *Proceedings of the IEEE-CS International Metrics Symposium*, pages 71–81, 1993.

[23] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, volume 19(5) of *ACM SIGPLAN Notices*, pages 177–184, 1984.

[24] K. Pan, S. Kim, and J. E. James Whitehead. Bug classification using program slicing metrics. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006)*, pages 31–40, 2006.

[25] M. Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, 1979.