ELSEVIER

The Journal of Systems and Software 54 (2000) 159–166

The Journal of Systems and Software

www.elsevier.com/locate/jss

Locality metrics and program physical structures

Kang Zhang [a,b,*], Narasimhaiah Gorla [c]

[a] Department of Computer Science, University of Texas at Dallas, Box 830688, MS EC31, Richardson, TX 75083-0688, USA
[b] State Key Laboratory for Novel Software Technology, Nanjing University, People's Republic of China
[c] Department of Computing, Hong Kong Polytechnic University, Hung Hom, Hong Kong, People's Republic of China

Received 24 November 1999; received in revised form 15 April 2000; accepted 15 May 2000

Abstract

Years of programming experience has convinced us that the physical structure of a program, such as the locations of the program's components, their calls, and the depth of nested calls, is important in determining how effective and efficient the program can be debugged and maintained. This paper introduces a new class of physical metrics, known as locality metric, that measures the relative positions of components in a program listing and reveals useful attributes that may affect programmer productivity. The placement of the components can be determined by a simple algorithm that is of polynomial time complexity. The paper compares the performance of the algorithm with that of an exhaustive search approach and also reports various characteristics of the locality metric based on the collected statistical data. The performance shows the feasibility of the algorithm and closeness of its output to the optimal result found by the exhaustive approach.

Keywords: Locality metric; Physical structure; Logical structure; Program component

1. Introduction

Billions of dollars are spent each year world-wide on software costs, a large portion of which is used for software maintenance (Gorla, 1991). A major reason for this is that the software quality is typically low and results in high maintenance overhead. Various software metrics have been used to quantitatively characterise essential features of programs. Structure metrics have been used to analyse the logical relationships among program components (Chidamber and Kemerer, 1994; Dharma, 1995; Gorla and Ramakrishnan, 1997; Henry and Kafura, 1981). Such logical structure metrics include coupling and cohesion complexity between program components (Dharma, 1995), internal complexity, positional complexity, and interfacing complexity of a component (Adamov and Richter, 1990), the span of control and its distribution (Gorla and Ramakrishnan, 1997), and object-oriented metrics for predicting fault-proneness (Basili et al., 1996; Chidamber and Kemerer, 1994). Research on analysing physical attributes of a program such as style characteristics has produced some interesting and useful results (Benander et al., 1990; Berry and Meekings, 1985; Gorla et al., 1990).

However, little attention has been paid to the physical layout of program components. It is obvious that a programmer's productivity and particularly debugging effort not only depend on the logical structure but also on the physical structure of the program. Programmers, in real life, go through program listings many times for debugging and maintenance purposes, particularly with large legacy programs. Easy-to-read programs lead to low debugging and maintenance effort. For example, consider a simple program consisting of three large components that are placed in the order of A, B, C. A includes at least one call to C and B has no calling relationship with A or C. It is apparent that during program debugging, the programmer has to flip over the B listing when inspecting the program code of A and C. This would waste the programmer's time but can be easily avoided if the program components are placed in an appropriate order. This paper proposes a measurement that characterises the placement ordering of components in a program, and reports our findings of the relationship between program components' placement and the measurement, and how a better measurement and thus better placement may be obtained.

* Corresponding author. Tel.: +1-972-883-6351; fax: +1-972-883-2349.
E-mail address: kzhang@utdallas.edu (K. Zhang).

0164-1212/00/$ - see front matter
PII: S0164-1212(00)00059-5

This paper introduces a new class of metrics, called *locality metric*, to quantify the physical structure of the above nature. A measurement makes sense only when it is associated with one or more models because experimentation models of measurement are essential for case studies and experiments for software engineering research (Pfleeger et al., 1997). The experimentation model of our research can be stated in the following aspects.

- *Problem domain.* The main objective of the present study is to devise a measurement that quantifies the placement ordering of program components. The factor that determines the ordering of a number of components is the calling relationships among the components and is measured by the locality metric. The ultimate aim is to associate the locality metric with the programmer's productivity.
- *Language paradigm.* We consider software development in any conventional imperative programming language that features function or procedure calls. We believe that our study and results are equally applicable to some other programming paradigms.
- *Hypotheses.* The program development platform is assumed to support a usual screen editor with the scrolling capability to view and edit a long program listing. Some special editors with structural support, such as a folding editor provided in the Occam Development Kit for transputers, are assumed not in use.
- *Method of experiment.* We run the implemented algorithms on a workstation to observe their real-time performance and collect various statistical data to analyse the characteristics of the locality metric.

The rest of the paper is organised as the following. Section 2 briefly compares logical and physical metrics, followed by Section 3 that introduces the concept of locality metrics through a simple example. Section 4 describes an algorithm for finding a near-optimal component placement and reports the initial experimental results. Section 5 discusses the impact of logical structures on the effectiveness of the locality metric. The paper is finally concluded by Section 6.

2. Logical and physical metrics

Logical metrics measure a program's functionality and correctness. Examples of logical metrics include span of control, cyclomatic complexity, etc. Physical metrics measure a program's physical structure which include ergonomic factors that affect the programmer productivity, such as relative positions of a component call and the corresponding component to be called. The two types of metrics are orthogonal and cannot be derived from one another. Combined logical and physical metrics provide a means for comprehensively measuring programmer productivity. Of the software development cycle, we are mostly concerned with three critical stages,

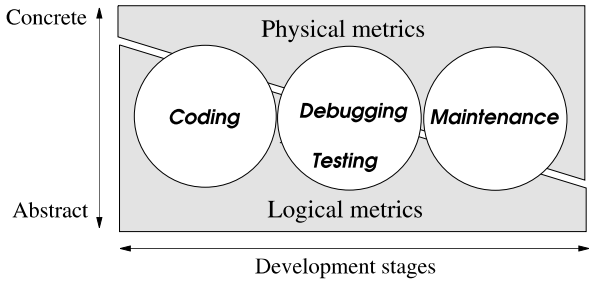

Fig. 1. Factors of programmer productivity.

i.e. *coding, debugging*, and *maintenance*. Different metrics play different roles during these development stages.

As depicted in Fig. 1, logical metrics are primarily used at the coding stage, where the logical structure guides the program construction through its data and control flows. Physical metrics, on the other hand, are useful in providing measurements necessary for software maintenance. Both types of metrics can aid program debugging, but in very different ways. The former assists logical debugging and error detection while the latter may help improve debugging efficiency. Since their roles are complementary, the combined use of logical and physical metrics should provide comprehensive measurements over the stages of coding, debugging, and maintenance.

3. Locality metrics

With years of programming experience, we observed that the physical locations of components and their calls, and the depth of nested calls are important factors that affect debugging and maintenance. These factors form a class of physical metrics, known as *locality metrics* (Gorla and Ramakrishnan, 1997). They measure relative positions of program components and reveal useful attributes that affect the programmer productivity. A locality metric (LM) is defined as the sum of the distances between all pairs of calling components and called component divided by the total number of calls:

$$\mathrm{LM} = \sum_i \sum_j (F_{ij} D_{ij}) / \sum_i \sum_j F_{ij} = D_{\mathrm{tot}} / \sum_i \sum_j F_{ij},$$

where $D_{ij} = | i - j |$ is the *distance* between calling component i and called component j, F_{ij} is the frequency of calls from i to j, $D_{\mathrm{tot}} = \sum_i \sum_j (F_{ij} D_{ij})$ is the *total distance* that includes all the calls between any pairs of components.

The above metric assumes that the program components are of about the same size. The assumption is unrealistic for most application programs. To take the program size into account, we introduce the concepts of *internal distance* and *external distance* below. If a calling statement is placed before the called component, it is a

forward calling statement. If after the called component, it is a *backward* calling statement. In the following discussion, $i < j$ implies that i is placed before j and $i + 1$ is the component placed immediately after i.

Definition 1. Internal distance. The *internal distance* D_{in_i} for a forward calling statement in component i is the number of lines of code between the statement and the end of i. The *internal distance* for a backward calling statement in component i is the number of lines of code between the beginning of i and the statement.

Definition 2. External distance. The *external distance* $D_{\text{ex}_{ij}}$ between a calling component i and a called component j is the sum of lines of code in the components between i and j. Therefore, for a forward call ($i < j$)

$$D_{\text{ex}_{ij}} = \sum_{k=i+1}^{j-1} L_k$$

and for a backward call ($i > j$)

$$D_{\text{ex}_{ij}} = \sum_{k=j+1}^{i-1} L_k,$$

where L_k is the number of lines in component k.

Hence, when the distance D_{ij} is measured by lines of code, it equals $D_{\text{in}_i} + D_{\text{ex}_{ij}}$. The *total distance* D_{tot} for all the calls, taking different sized components into account, is the total number of calls times the sum of internal and external distances for all the calls:

$$D_{\text{tot}} = \sum_i \sum_j F_{ij}\left(D_{\text{in}_i} + D_{\text{ex}_{ij}}\right),$$

where F_{ij} is the frequency of calls from i to j. To ease and simplify the discussion of the following examples, the distance between any two components is still measured by the number components between them, rather than by the number of lines between the calling statement and called component. The principle and approach described in the remaining part of the paper apply to real programs when measuring them in either lines of code or number of components.

The main aim of introducing locality metrics is to guide the programmer in placing or re-arranging program components in such a way that the programmer's debugging and maintenance efforts can be minimised. Consider the logical structure of a program shown in Fig. 2 which reveals the relationships between calling and called components (modules or procedures). The components are identified by their numbers in circles and 1 represents the main component. The number next to an arc represents the calling frequency between the two components connected by the arc.

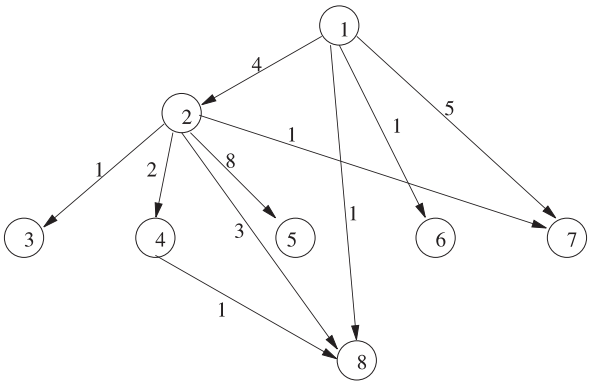

Fig. 2. An example of logical structure of a program.

Assuming that the physical structure of the program, i.e. the ordering of the components in the program listing is

1 2 3 4 5 6 7 8,

the 8×8 matrix representing the calling structure is shown in Table 1. In the matrix, the rows and columns are marked by the component numbers as their identifiers, each matrix element represents the calling frequency between the two components at the corresponding row and column, and M is an integer larger than the largest calling frequence in the matrix. The calling direction is ignored, so that A calling B is regarded the same as B calling A. Thus the calling matrix is always symmetric. The locality metric for the program is calculated as shown in Table 2, where "Distance 1" denotes the number of components between the calling and called components, multiplied by the calling frequency.

The locality metric is 3.778, obtained by dividing the total distance (i.e., 102) by the total number of calls (i.e., 27 – calling frequency total). If the program components have the following different ordering

1 7 8 2 4 3 6 5,

the locality metric is then calculated as in the right-most column, denoted by "Distance 2", of the same table. The locality metric is 2.519, which is lower than the former program structure. The lower the locality metric

Table 1
Original calling matrix of the example program

	1	2	3	4	5	6	7	8
1	M	4	0	0	0	1	5	1
2	4	M	1	2	8	0	1	3
3	0	1	M	0	0	0	0	0
4	0	2	0	M	0	0	0	1
5	0	8	0	0	M	0	0	0
6	1	0	0	0	0	M	0	0
7	5	1	0	0	0	0	M	0
8	1	3	0	1	0	0	0	M

Table 2
The locality metrics of the example program

Calling component	Called component	Calling frequency	Distance 1	Distance 2
1	7	5	6×5	1×5
1	8	1	7×1	2×1
1	2	4	1×4	3×4
1	6	1	5×1	6×1
2	7	1	5×1	2×1
2	3	1	1×1	2×1
2	8	3	6×3	1×3
2	4	2	2×2	1×2
2	5	8	3×8	4×8
4	8	1	4×1	2×1
Total Distance			102	68
Locality Metric			3.778	2.519

is, the closer the related program components (i.e., the ones having calling relationships) are placed in the program listing. The second program structure should result in a lower debugging and maintenance overhead than the first program structure, given the same logical structure. This is because the programmer spent less time in inspecting related program components which are placed together or close to each other.

4. Component placement

In order to decide the placement of a program's components that gives a minimal locality metric, we adapt the bond energy algorithm (BEA) (McCormick et al., 1972) to group interrelated components together.

4.1. Bond energy algorithm

The *bond energy algorithm* (BEA), proposed by McCormick et al. (1972), is a cluster-analysis method for identifying natural variable groups and clusters in complex data arrays. It introduces the concept of *measure of effectiveness* (ME), aiming at maximising the summed bond energy (i.e., ME) over all row and column permutations of an input array. That is, find

$$\max(\mathrm{ME}) = \max\left\{ \sum_{i=1}^{M} \sum_{j=1}^{N} a_{i,j}[a_{i,j-1} + a_{i,j+1} + a_{i-1,j} + a_{i+1,j}] \right\}$$

for all $N!M!$ permutations (with the convention $a_{0,j} = a_{M+1,j} = a_{i,0} = a_{i,N+1} = 0$).

For problems that decompose into subproblems that are non-interacting or interact only through a few well-defined interface variables, the BEA is useful in identifying their subproblems and interfaces. Applications of the BEA include grouping various types of machinery by their functions, grouping warehouses by the area they service, etc.

According to the calling relationships and frequencies of program components, we adapt the BEA to decide the components placement. We use an $N \times N$ symmetric array, whose rows and columns identify the N program components and each array element is the calling frequency between the two components at the corresponding row and column. Since the array is represented as a symmetric matrix, we need only to find a row (or column) permutation that creates the strongest 'bond energy' by driving the larger array elements together. This is achieved by calculating the measure of effectiveness (ME) for each permutation by

$$\mathrm{ME} = \sum_{i=1}^{N} \sum_{j=1}^{N} F_{i,j}(F_{i,j-1} + F_{i,j+1}),$$

where $F_{i,j}$ is the calling frequency between components i and j. The permutation that gives the maximal ME represents a desirable component placement, which is near-optimal as discussed in Section 4.3. Since the initial matrix is symmetric, the algorithm can operate on either rows or columns and generate the same result as described below.

Given a program of N components, assuming to operate on rows:

1. Create an $N \times N$ symmetric matrix $InitMatrix = [F_{i,j}]$ such that $F_{i,j}(= F_{j,i})$ is the calling frequency between components i and j ($i \neq j$), and $F_{i,i}$ is an integer larger than the maximal value of the matrix elements.
2. Select a row arbitrarily (say, the first row) from *InitMatrix* to be placed in a new matrix. Set $i = 1$.
3. Try to place individually each of the remaining $N - i$ rows in each of the $i + 1$ possible positions, and compute each row's contribution to the ME. Place the row in the position that gives the largest incremental ME. Increment i by 1 and repeat this step until $i = N$.
4. The rows of the resulting new matrix (*FinalMatrix*) gives the relative positions of the program components.

Table 3
Resulting matrix with the ordering of 34852176

	1	2	3	4	5	6	7	8
3	0	1	M	0	0	0	0	0
4	0	2	0	M	0	0	0	1
8	1	3	0	1	0	0	0	M
5	0	8	0	0	M	0	0	0
2	4	M	1	2	8	0	1	3
1	M	4	0	0	0	1	5	1
7	5	1	0	0	0	0	M	0
6	1	0	0	0	0	M	0	0

Consider again the example program in Fig. 2. The above algorithm reorganises the rows to generate the matrix in Table 3, whose total distance is 41 and the locality metric is 1.518, smaller than those of the ordering structures in Table 2. The resulting order of the components is

3 4 8 5 2 1 7 6.

4.2. Performance

Step 3 of the above algorithm is the core of the algorithm and is the most time-consuming part. Its implementation in C is shown below:

```
MaxME = 0;
for (i=1; i<N; i++) {
  for (j=0; j<=i; j++) {
   InsertRow(InitMatrix[i], TempMatrix, j,
   i);
   if (CompME(i) > MaxME) {
    MaxME = CompME(i);
    pos = j;
   }
  }
  InsertRow(InitMatrix[i],    FinalMatrix,
  pos, i);
}
```

where `InsertRow(F[i], TempMatrix, j, i)` inserts the *i*th row of the matrix `InitMatrix` into the matrix `TempMatrix[i][N]` at the position `j`. `CompME` calculates the measure of effectiveness using the above equation. The program involves four nested loops of the size up to N. Therefore, the time complexity of the algorithm is $O(N^4)$, which is quite acceptable. According to McCormick et al. (1972), the final groupings of the resulting matrices are insensitive to how the initial row (column) is selected (Step 2 of the algorithm) and their resulting MEs have only fractional differences.

How is the above algorithm compared with the exhaustive approach which checks the locality metrics for all row (column) permutations and finds the optimal row (column) ordering based on the smallest locality metric? An $N \times N$ matrix has $N!$ row permutations. The time complexity of $O(N!)$ is obviously unacceptable when N is reasonably large.

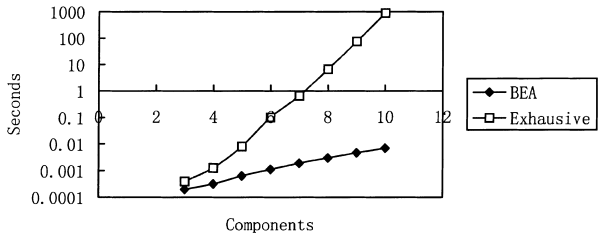

Fig. 3. Performance comparison of the BEA and exhaustive search (mixed calling frequencies).

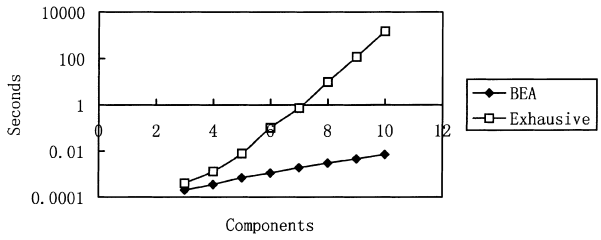

Fig. 4. Performance comparison of the BEA and exhaustive search (uniform calling frequencies).

To compare the actual performance of the exhaustive approach with that of the BEA, we ran the implementations for both approaches. The running times on a SPARCstation 20 for the two approaches when $N = 3$–10 are shown in Figs. 3 and 4. The calling patterns are the same for both figures, except that Fig. 3 depicts the performance for mixed calling frequencies while Fig. 4 depicts the performance for uniform calling frequencies (all equal to 1, i.e., for each calling frequency in the former, if $F^1_{i,j} \neq 0, F^2_{i,j} = 1$ is the frequency in the latter). With the former, the exhaustive approach for $N = 11$ took more than 3 h while the BEA needed less than 0.1 s. With the latter, the exhaustive approach for $N = 11$ took more than 10 h.

4.3. Usefulness of BEA and other characteristics

To see whether the time is worth spending on finding the optimal locality metric using the exhaustive approach, we also compare the locality metrics recommended by the BEA and the optimal locality metrics found by the exhaustive approach, with mixed calling frequencies as listed in Table 4 and with uniform calling frequencies as in Table 5. The mixed calling frequencies are generated randomly and the uniform calling frequencies are obtained by replacing every non-zero frequency with 1. The calling structure of each test case in Table 4 is the same as the calling structure of the corresponding test case in Table 5. The tables show that the differences between the locality metrics found by the two approaches are insignificant – maximally 18% (Table 4: five components) for mixed calling frequencies and 17% (Table 5: nine components) for uniform calling

Table 4
Original, BEA generated and the optimal locality metrics (mixed calling frequencies)

No. of components	3	4	5	6	7	8	9	10	11
Original	2.000	2.455	1.411	3.079	2.880	3.778	3.073	4.520	4.300
BEA generated	1.000	1.000	1.311	1.333	1.478	1.518	1.185	1.530	1.285
Optimal	1.000	1.000	1.078	1.325	1.402	1.481	1.129	1.498	1.206

Table 5
Original, BEA generated and the optimal locality metrics (uniform calling frequencies)

No. of components	3	4	5	6	7	8	9	10
Original	2.000	1.667	2.167	2.182	2.571	4.000	4.125	3.545
BEA generated	1.000	1.000	1.667	1.818	2.071	1.900	2.625	2.545
Optimal	1.000	1.000	1.500	1.818	2.000	1.800	2.188	2.409

frequencies. However, only the BEA approach is feasible for programs with a large number of components due to its polynomial running time.

Further characteristics of locality metrics of different component orderings can be observed as the following:

- With varied calling frequencies and irregular calling structures, the differences between the locality metrics of randomly ordered components and those generated by the BEA are larger than those with more uniform calling frequencies.
- A general trend is that as the number of program components (i.e., the size of the calling matrix) or the complexity of the calling structure increases, the locality metric increases slowly.
- The more the program components (i.e., the bigger the matrix size is), the more impact the calling frequencies may have on the locality metric. With a small number of program components, changing calling frequencies without changing the calling pattern may not change the optimal locality metric. The BEA may also generate the same locality metric and thus recommend the same components ordering as the optimal one.
- The components ordering of the optimal locality metric and that recommended by the BEA are very similar. In most cases, the differences lie in the ordering of only two components, occasionally three.

5. Impact of logical structures

We have assumed that the program's logical structure has been determined and cannot be changed, while the program's components may be reordered in the program listing. If, however, the calling relationships, i.e., logical structures, can be changed with a minimal effort, how is the locality metric to be effectively applied? This section discusses some possible impacts of logical structures on locality metrics.

Consider the following two logical structures: a component calls a number of other components directly (Fig. 5(A)) and the same number of components form a calling chain with nested calls (Fig. 5(B)). Assuming the same physical structure, i.e., the five components are placed in the order of 1, 2, 3, 4, 5 and the calling frequencies all equaling 1, the locality metric for Program A is $(1+2+3+4)/4 = 2.5$, while that for Program B is $(1+1+1+1)/4 = 1$. The locality metric of Program B is significantly lower than the locality metric of Program A. Our previous investigation has shown that the program development time increases with an increase in the levels of nesting (Gorla and Ramakrishnan, 1997). It is obvious that Program B is much harder to debug and maintain than Program A. Therefore, the above locality metric alone is insufficient in comparing programs of different logical structures for their debugging and maintenance costs. Its definition needs to be revised to cater for the situation when logical structures are allowed to change.

The call depth is obviously an important logical structure that directly affects the programmer's debugging and maintenance effort. To take the call depth into

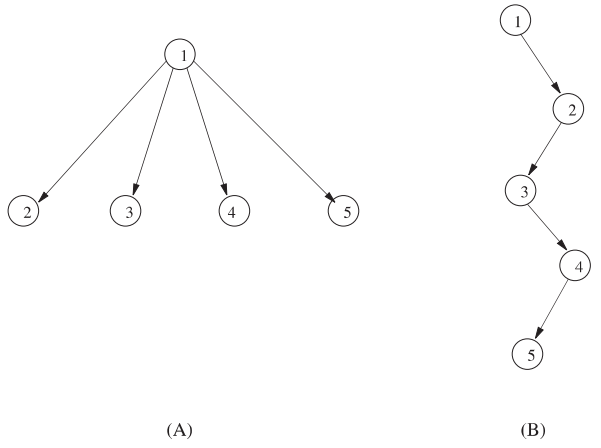

Fig. 5. Two different calling structures.

account, we introduce the concept of program height H, which is defined as the maximum of the levels of all the component calls in a program

$$H = \max_i \left(\max_j L_{ij} \right) - 1,$$

where L_{ij} is the maximal number of components involved in the calling sequence from the first calling component i to the last called component j. Therefore, the program height is the maximal number of arcs between any two components in the program. For instance, in the example program shown in Fig. 2, $L_{18} = 4$ involves the longest calling sequence from component 1 to component 8, and thus $H = 3$.

The locality metric can be redefined as the following

$$H \sum_i \sum_j D_{\mathrm{tot}_{ij}} / \sum_i \sum_j F_{ij},$$

where H is the program height. The locality metric for Program A is now $2.5 \times 1 = 2.5$ and that for Program B is $1 \times 4 = 4$.

Another factor associated with logical structures to be considered in improving programmer productivity is the component in-line technique, i.e., placing the called component into the calling component. We call such an action of program modification *flattening*. We only consider the simplest flattening, i.e., to flatten those small components, each of which is called by only one other component and is placed far away from its calling component. The aim of such flattening is to minimise the locality metric.

Considering again the same example program structure in Fig. 2, our algorithm finds the near optimal component ordering of 34852176. It can be seen from this ordering and from Fig. 2 that the distance between components 2 and 3 is 4, while component 2 calls component 3 only once. So the calling relationship between 2 and 3 is a good candidate to be removed (i.e., flattened). After putting component 3 in-line of component 2, the locality metric is further reduced, from 1.518 to 1.423. The resulting ordering remains the same with component 3 removed, that is, 4852176. It should be pointed out, however, that flattening may reduce the component cohesion so that debugging a component involving a flattened component may cost a little more time than debugging the same component without flattening.

6. Conclusion

Motivated by the recognition of the importance of a program's physical structure, the paper has introduced an important type of physical metrics, known as locality metric. The locality metric measures the ordering and relative positions of components in a program. The adapted bond energy algorithm can guide the placement of program components so that the relevant components (having calling relationships) are placed together or closer to each other in the program listing.

Our experiments with the implementation of the BEA and that of exhaustive search for optimal ordering have demonstrated the feasibility, usefulness, and efficiency of the BEA in finding a near-optimal ordering. The experiments have also revealed some characteristics of the algorithm and how it is affected by different calling profiles (frequencies, regularity, etc).

With the algorithm implemented in a programming tool or environment, the placement of program components can be easily automated so that a newly coded program or an existing legacy program can be properly reorganized prior to the debugging/maintenance activities. A program visualisation tool that supports the construction and display of call trees could be naturally extended with the capability of automatic re-ordering of program components, possibly with visualisation of the re-ordering process.

The future work includes the empirical study of the locality metric to find how useful it is in software development process, particularly in reducing the debugging and maintenance effort. Such empirical studies will also verify the impact of logical structures and the sizes of program components, when considering calling distances. Our previous investigation has shown that the optimal component size is about 50–60 lines of code (Gorla et al., 1990). This is consistent with the physical constraints such as the screen size. We will further investigate the effectiveness of the locality metric with these physical constraints.

Acknowledgements

We would like to thank the reviewers for their useful comments that helped us to improve the final presentation. The work was supported by a Hong Kong Polytechnic University research grant under grant number YB10.

References

Adamov, R., Richter, L. 1990. A proposal for measuring the structural complexity of programs. J. Systems and Software 55–70.

Basili, V.R., Briand, L.C., Melo, W.L., 1996. A validation of Object-Oriented Design Metrics as quality indicators. IEEE Trans. Software Engrg. 22 (10), 751–761.

Benander, B., Gorla, N., Benander, A., 1990. An empirical study of the use of the GOTO statement. J. Systems and Software.

Berry, R.E., Meekings, B.A.E., 1985. A style analysis of C programs. Comm. of the ACM 28 (1), 80–88.

Chidamber, S.R., Kemerer, C.F., 1994. A metrics suite for object-oriented design. IEEE Trans. Software Engrg. 20 (6), 476–493.

Dharma, H., 1995. Quantitative Models of Cohesion and Coupling in Software. J. Systems and Software 65–74.

Gorla, N., 1991. Techniques for application software maintenance. Inform. Software Technol. 65–73.

Gorla, N., Ramakrishnan, R., 1997. Effect of software structure attributes on software development productivity. J. Systems and Software 191–199.

Gorla, N., Benander, A., Benander, B., 1990. Debugging effort estimation using software metrics. IEEE Trans. Software Engrg. 16 (2), 223–231.

Henry, S., Kafura, D., 1981. Software structure metrics based on information flow. IEEE Trans. Software Engrg. 1981.

McCormick, W.T., Sweitzer, P.J., White, T.W., 1972. Problem decomposition and data reorganisation by a clustering technique. Oper. Res., (September–October) 993–1009.

Pfleeger, S.L., Jeffery, R., Curtis, B., Kitchenham, B., 1997. Status Report on Software Measurement. IEEE Software, (March/April) 33–43.