A cohesion metric approach to dividing source code into functional segments to improve maintainability

Norihiro Yoshida, Masataka Kinoshita, Hajimu Iida
Graduate School of Information Science, Nara Institute of Science and Technology
8916-5 Takayama, Ikoma, Nara 630-0192 Japan
Email: yoshida@is.naist.jp, iida@itc.naist.jp

Abstract—**During software maintenance, understanding source code is one of time-consuming activities. Good programing practice suggests that programmers should insert blank lines to divide source code into functional segments, and a comment at the beginning of each functional segment. Those help developers to understand functional segmentation of source code, such as start and end points of each functional segment. Unfortunately, programmers do not always follow this practice. In this paper, we propose an approach to dividing source code into functional segments. Our approach uses cohesion metric for code portion to identify start and end points of each functional segment.**

I. Introduction

It has been pointed out that, for software maintenance operations, an enormous time and cost is involved in the understanding of the source code [1][2]. Even in the case of source code created by maintainers themselves, if they have not handled it for a prolonged period, for example, or other developers have made alterations, a long time may then be required for its understanding. The source code is an accurate representation of the system but is too vast and detailed for the developer to comprehend. For that reason, a bottom-up approach to understanding the whole thing, that is, reading one line at a time from the first line of each source file onwards, is a very difficult task. Therefore, the maintenance person in charge generally understands the source code by a kind of top-down approach starting with an understanding of the system, and subsequently the modules and functions, rather than reading through every line from the top line of each function. This also similarly applies for a kind of top-down understanding of functions themselves. Following an overall understanding starting with the initial comments, function name, and so on, and then identifying the locations where each individual feature of that function is implemented, a final understanding of the code fragments (regions of uninterrupted source code) which implement each of those features is obtained.

However, when modularization is not done properly, the top-down understanding of functions is difficult [2]. Also, it similarly results in difficulties in top-down understanding if either documentation or source code comments are lacking, and the features of a function cannot be associated with the code fragments inside it [3]. In this study, with the objective of aiding function comprehension, we propose a technique to extract sets of code fragments which realise the same features within a function by using cohesion metrics. The users of this technique are assumed to be developers who have understood the intended function at least once and not those developers who are studying the function for the first time. Firstly, in this study, in order for a set of code fragments to realise a specific feature, and in order to measure the extent to which they work together, we define cohesion metrics designed for sets of code fragments. Then, those sets of code fragments which have a cohesion metric above the threshold are detected and presented as functionalities. Because functionality can be extracted automatically from just the source code, this approach can be used even in such cases when test cases are inadequate or there is no runtime environment.

Additionally, in the application experiments, we have applied a tool which implements the proposed technique for actual software. These results show that, if the cohesion metrics threshold is set appropriately, the majority of the functionalities presented by this tool agree with those found by hand.

II. Cohesion Metrics

In general, cohesion is a measure which expresses, in order for each of the constituent parts within a module to realise a specific feature, the extent to which they work together [4]. The cohesion takes a high value for a module which implements a single feature, and conversely, a low value if it implements multiple features which are not related. As typical cohesion metrics, those proposed by Chidamber et al. can be cited [5], the LCOM (Lack Of Cohesion in Methods) class which express the lack of cohesive properties. LCOM considers attributes as data elements and methods as functional elements, and expresses the degree of coordination between them.

Miyake et al. devised the COB (Cohesion of Blocks) cohesion metrics which express cohesion in methods [6]. COB considers the variables used in the method as data elements, and blocks of code as functional elements, and expresses the degree of coordination between them. COB is defined by the formula below which is based on the number of blocks from which each accessible variable in the method can be accessed.

$$COB = \frac{1}{b}\frac{1}{v}\sum_{j}^{v}\mu(V_j) \qquad (0 \leq COB \leq 1)$$

V_j	: j-th accessible variable in the method
v	: # of variables that can be accessed in the method
b	: # of blocks in the method
$\mu(V_j)$	: # of blocks that access Vj

III. Proposed technique

In this study, with the objective of aiding the understanding of functions, we propose a technique to extract from within a function, sets of the code fragments which realise different features. In this study, the source code and the features are defined as follows.

- a single piece of software source code S is divided into a set of code fragments $CF = \{cf_0, cf_1, \ldots, cf_m\}$.
- the source code S realises a set of features $F = \{f_0, f_1, \ldots, f_n\}$
- each feature in F is realised by coordinating more than one of the code fragments included within CF. However, the code fragments included in CF can belong to multiple features within F.

Using these definitions, the purpose of this study is to propose a technique to obtain the $Z_i \in CF$ (here, $0 \leq i \leq n$, $\bigcup_{i=0}^{n} Z_i = CF$) which realise an $f_i \in F$.

In this study, we define a cohesion metric for a code fragment set, and using that, identify the code fragment set which works together to realise a single feature, and present that as a functionality. From section III-A onwards, we define cohesion metrics for code fragment sets, and in section III-B, we propose a method to extract functionalities using those cohesion metrics.

A. Cohesion metrics designed for code fragment sets

We propose cohesion metrics which are designed for sets of code fragments to obtain the set of code fragments which work together to realise a given feature. Firstly, we extend the COB cohesion metrics designed for methods to define the cohesion metrics COCP for sets of code fragments. COCP regards variables as data elements, and code fragments as functional elements, and represents their degree of coordination. When any of these variables is accessed by one or more code fragments, for code fragment sets composed of 2 or more elements, COCP is represented by the following formula.

$$COCP = \frac{1}{p}\frac{1}{v}\sum_{j}^{v}\mu(V_j) \qquad (0 < COCP \leq 1)$$

V_j	: j-th accessible variable in the set
v	: # of variables that can be accessed in the set
p	: # of code fragments
$\mu(V_j)$	: # of code fragments that access Vj

COCP ranges in value between 0 and 1, but it does not take small values when there are only a few code fragments. When the number of code fragments is 2, for example, COCP takes on a value of 0.5 even for a combination of code fragments which are not coordinated at all. In this way, COCP is very sensitive to the number of code fragments.

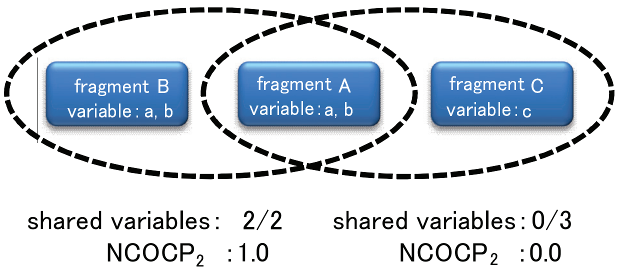


Fig. 1. Example of Calculation of NCOCP$_2$

We therefore define $NCOCP_2$ which has been normalized for code fragment number. This normalization was performed in the same way as LCOM*, which is normalized by the number of LCOM methods [7]. $NCOCP_2$ is defined by the following formula using COCP and the code fragment number p.

$$NCOCP_2 = \frac{p - 1/COCP}{p-1} \qquad (0 \leq NCOCP_2 \leq 1)$$

The proposed technique, with the use of $NCOCP_2$, identifies a set of code fragments which are coordinating through variables to realise a given feature, and presents this to the developer. It is easy for developers using this proposed technique to associate the features in a function with code fragments within that function, and it is therefore believed to reduce the costs involved in program comprehension. Figure 1 shows an example calculation of $NCOCP_2$. Because there are some variables shared between code fragment A and code fragment B, $NCOCP_2$ between them is a high value. Conversely, between code fragments A and C, $NCOCP_2$ is a low value.

B. A cohesion-based technique for extracting functionalities

1) Step 1: syntax analysis: This step divides the source code into code fragments based on the program syntax to create a unique syntax tree comprising the program syntax as nodes, and the code fragments as leaves. Next, we illustrate this specific procedure.

(1) **Syntax tree initialization**
The whole source code is taken as the initial processing range. The root node representing the whole source code is added to the syntax tree, and taken as the current node.

(2) **Block search**
Blocks are searched for within the current processing range scanning from the first line.

(3-a) **On encountering a block**
When a block is found by (2), the source code is divided into 3, “from the first line of the processing

range to the first line of the block", "from the first line of the block to the end of the block", and "from the end of the block to the end of the processing range". Steps (2) onwards are applied recursively for each range around the block. Then, after adding the block which was found as a node onto the current node of the syntax tree, and making that the next current node, steps (2) onwards are also applied for the range in the block interior itself. The recursive process is continued until (3-b) is reached.

(3-b) **On completion of scanning the processing range** If scanning of the processing range with step (2) is completed without finding program syntax, the code fragments in that range are added to the current node of the syntax tree as a leaf. Also, information is provided to the code fragments about the variables which are accessed in that range.

(4) **Output of the syntax tree** When the entire source code range has been split and added to the syntax tree using steps (2) and (3), the syntax tree is then output.

Figures 2 and 3 show examples of creating a syntax tree from the source code for some dummy programming language. (a) in Figure 2 is an example of splitting up the source code into simple blocks. If the syntax is nested, it is recursively split up as in (b). (c) in Figure 3 is an example of splitting due to an if-else statement. If the source code is split up according to the control syntax, it is divided into the source code around the syntax, and then such things as conditional statements, execution statements, and update statements. (d) is an example of splitting with a call to an expandable function, but excludes such as external APIs or recursive functions. These kind of functions and methods which perform in-line expansion are processed in the same way as blocks. For that reason, functions and the like which are called in several places may belong to multiple features.

2) Step 2: extraction of functional elements: Steps 2 and 3 intended to match up, from among the group of code fragments obtained in Step 1, those thought to be working in cooperation. The syntax tree is taken into consideration in step 2, so that, more so than in step 3, code fragments with a syntactically close positional relationship will be matched up preferentially. Whether or not code fragments are working in cooperation is determined if the value of the cohesion metric defined in section III-A exceeds the threshold or not. As for the specific process, for each node on the syntax tree obtained in step 1, the values of the cohesion metric is calculated for the set of code fragments which belong to it. Given nodes and the nodes which belong to them for which the metrics values are above the threshold, the highest level nodes are thus located as the functional elements. This process is carried out to find those blocks and functions, in units of program syntax, which are working in conjunction over as broad a range as possible, and extract them as functional elements.

Figure 4 shows an example of extracting functionalities. The node number is at the top-left of the node. For example,

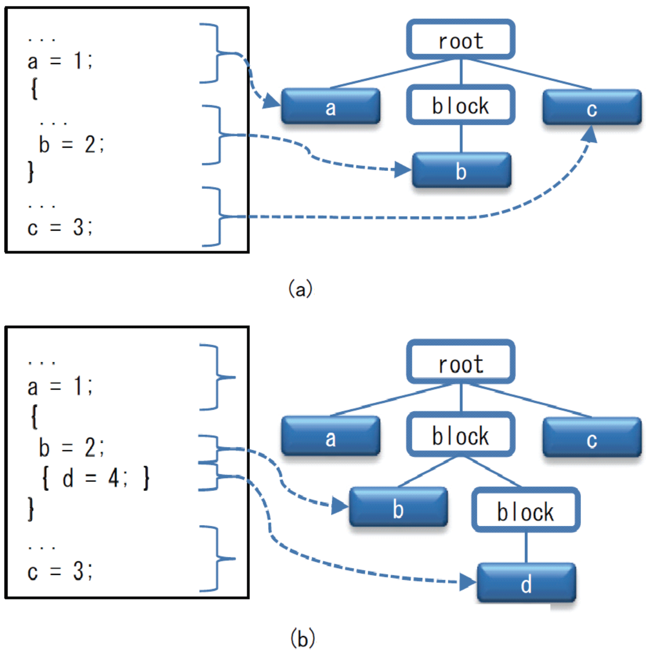

```
. . .
a = 1;
{
 . . .
 b = 2;
}
. . .
c = 3;
```
```
. . .
a = 1;
{
 b = 2;
 { d = 4; }
}
. . .
c = 3;
```

Fig. 2. Construction of Syntax Tree (1)

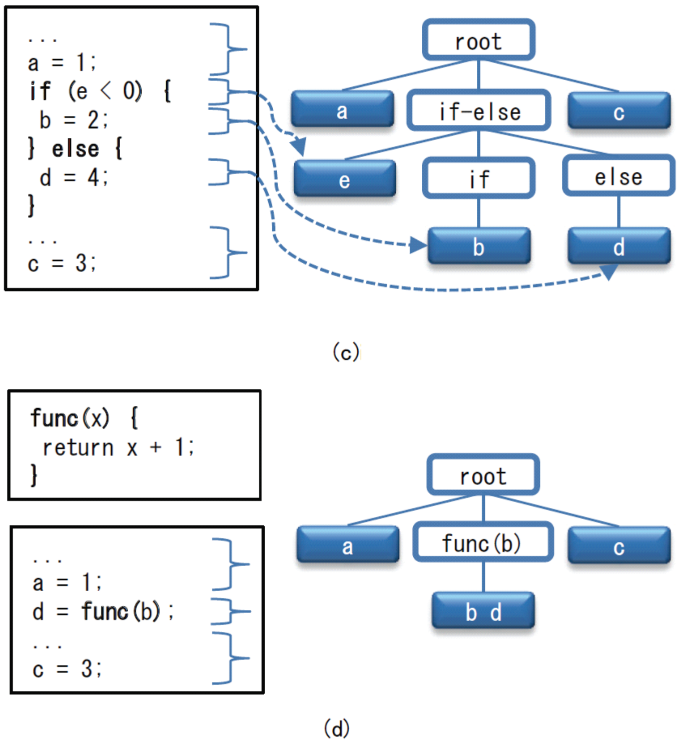

```
. . .
a = 1;
if (e < 0) {
 b = 2;
} else {
 d = 4;
}
. . .
c = 3;
```
```
func(x) {
 return x + 1;
}
```
```
. . .
a = 1;
d = func(b);
. . .
c = 3;
```

Fig. 3. Construction of Syntax Tree (2)

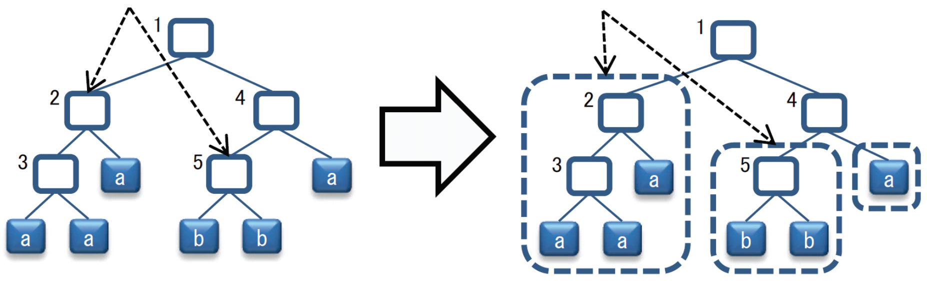


Fig. 4. Example of extracting functional elements

from the tendency of variable access in the code fragments, cohesion is high for the set of code fragments belonging to nodes 2, 3 and 5, whereas the cohesion is low for the set of code fragments belonging to nodes 1 and 4, and out of the nodes 2, 3 and 5, because node 3 is the lower level node to node 2, the remaining nodes 2 and 5 are functional elements. Also, code fragments that do not belong to either of node 2 or node 5 are single functional elements.

3) Step 3: extracting functionalities: In step 3, combinations of functional elements are found which have high cohesion, and functionalities are then extracted. Unlike step 2, since the positional relationship between matched up functional elements is not considered, even if the region belonging to a single feature is dispersed over positions separated throughout the source code, it can be extracted as a single functionality. A full search is done for combinations of functional elements to provide an evaluation value for cohesion.

IV. Case Study

We verified that the functionalities presented by the proposed technique agreed with those found manually by a developer. The Chem3D3 software for analyzing protein structure information was chosen as an application subject. Chem3D3 has been developed in C# and consists of 3641 lines, 70 classes, and 600 methods. When asked to put together a correct set of functionalities by hand, the developer in charge of the whole source code for Chem3D3 classified the entire source code into 80 features. It can be noted that with this classification, there were also code fragments classified into multiple features. The concordance rate for the functionalities presented by the proposed technique is expressed by the following, if we assign those due to the proposed method and those identified by the developer as A and B respectively, where spaces and tab characters have been removed from any character strings.

$$ratio(A,B) = \frac{2 \times |A \cap B|}{|A| + |B|}$$

Here, $A \cap B$ represents the common part between character strings A and B. When we applied the tool implementing the proposed technique to Chem3D3 and calculated the concordance rate of the functionalities presented with those identified by the developer, the results in Table 1 were obtained. The tool accepts source files written in C#, and calculates the location (i.e., begin and end lines) of each functional segment in given source files using proposed method.

When the threshold value for $NCOCP_2$ was set to 0.50, the average value, maximum value, and minimum value for the concordance rate were all the highest. In addition, when the threshold value for $NCOCP_2$ was set to 0.50, 51 functionalities were extracted, which was the closest to the 80 features found by the developer. For this reason, as far as application to Chem3D3 is concerned, a threshold value set to 0.50 can yield potential functionalities with high precision, and be considered an effective aid in source code comprehension. Figure 5 shows an example of the functionalities when setting the threshold value for $NCOCP_2$ to 0.50. This example includes 3 types of functionalities and agreed completely with the results found by the developer. The reading function for protein straddles 3 code fragments but could be extracted as a single functionality. The reason that a single functionality can be extracted even if it straddles multiple code fragments like that, is not because just a single code fragment is presented as a functionality, but because, as explained in section III-B, functional elements consisting of single code fragments are combined and the functionality presented.

Also, the identification function for protein structure is a 1-line method call statement, which is a single functionality. Since in-line expansion is performed as described in section III-B, the proposed technique detects functionalities only after in-line expansion by the method call in this example. Because the in-line expansion was not actually done by the proposed technique, due to the fact that the method call statement included in the identification function for protein structure shares its identifier ‘protein’ with other statements, the possibility exists that it is included in other functionalities.

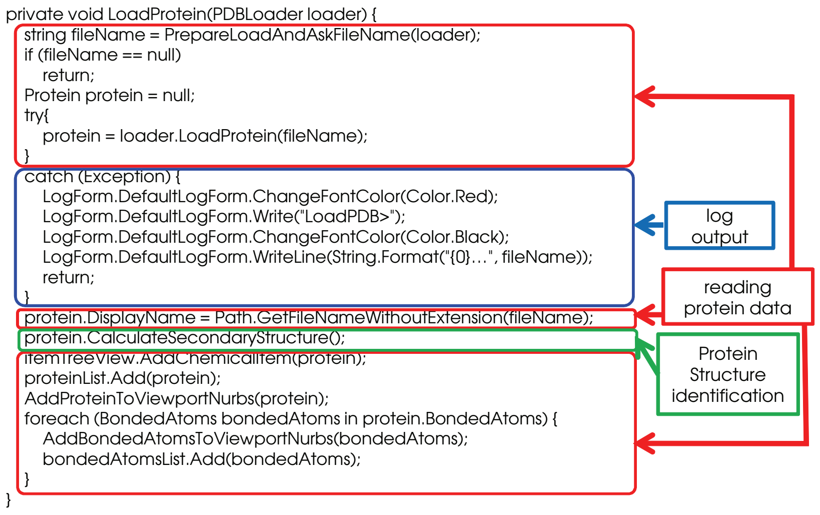
V. Related Work

Wang et al. [8] proposed an automatic blank insertion tool SEGMENT based on several heuristics to improve the readability of methods. We aimed to support software maintenance activity, and proposed a method to extract functional segments based on a cohesion metric $NCOCP_2$.

So far, several studies have been done on cohesion metrics based on program slicing[9][10][11][12]. Those metrics are promising to extract functional segments from source code, however, the computational costs of them are relatively higher

TABLE I
RESULT

$NCOCP_2$	# functionalities	concordance rate (ave.)	concordance rate (max)	concordance rate(min)
0.25	24	0.55	0.84	0.32
0.50	51	0.81	1.00	0.61
0.75	215	0.32	0.90	0.05



```
private void LoadProtein(PDBLoader loader) {
  string fileName = PrepareLoadAndAskFileName(loader);
  if (fileName == null)
     return;
  Protein protein = null;
  try{
     protein = loader.LoadProtein(fileName);
  }
  catch (Exception) {
     LogForm.DefaultLogForm.ChangeFontColor(Color.Red);
     LogForm.DefaultLogForm.Write("LoadPDB>");
     LogForm.DefaultLogForm.ChangeFontColor(Color.Black);
     LogForm.DefaultLogForm.WriteLine(String.Format("{0}...", fileName));
     return;
  }
  protein.DisplayName = Path.GetFileNameWithoutExtension(fileName);
  protein.CalculateSecondaryStructure();
  itemTreeView.AddChemicalItem(protein);
  proteinList.Add(protein);
  AddProteinToViewportNurbs(protein);
  foreach (BondedAtoms bondedAtoms in protein.BondedAtoms) {
     AddBondedAtomsToViewportNurbs(bondedAtoms);
     bondedAtomsList.Add(bondedAtoms);
  }
}
```

Fig. 5. Example of functionality

than proposed metric $NCOCP_2$. So, cost-cutting methods need to be considered.

Cohesion discussion can be extended also to other factors, not only the shared usage of variables to realize a feature. For instance, Counsell et al. [13] report with evidences that classes in object-oriented systems including more comments tend to be more cohesive. The proposed method is promising to be improved by incorporating other factors of cohesion (e.g. the number of comments).

VI. Concluding Remarks and Future Challenges

In this study, we have set out to provide support in the comprehension of functions, and proposed a technique to extract sets of code fragments which realise the same features within a function by making use of cohesion metrics. With the results of the application experiment, we were able to confirm that, by appropriately setting the threshold value for $NCOCP_2$, the majority of the functionalities presented by this method agreed with the results of a developer finding these features manually.

In the future, it is necessary to validate $NCOCP_2$ theoretically and establish it as a decision methodology common to languages and domains by applying it to other source codes. Future work should also include the proposal of a method to add feature names to the functionalities extracted by proposed method. Methods can be envisaged such as using a dictionary prepared in advance as in the 'Concept Assignment' proposed by Gold et al. [14], or the application of natural language processing technology so that keywords can be automatically extracted from code fragment sets.

Acknowledgment

We would like to thank Mr. Kenji Fujiwara and Mr. Rikichi Hirayama of Nara Institute of Science and Technology for their contributions to this work. This work is being conducted as a part of the StagE project. This work was also supported by Japan Society for the Promotion of Science, Grant-in-Aid for Scientific Research (No.22500027), and Grant-in-Aid for Research Activity start-up (No.22800040).

References

[1] L. E. Deimel, Jr., "The uses of program reading," *SIGCSE Bull.*, vol. 17, pp. 5–14, June 1985.
[2] D. R. Raymond, "Reading source code," in *Proc. CASCON '91*, Oct. 1991, pp. 3–16.
[3] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *IEEE Trans. Softw. Eng.*, vol. 29, pp. 210–224, 2003.
[4] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Systems Journal*, vol. 13, no. 2, pp. 115–139, 1974.
[5] S. Chidamber and C. Kemerer, "A metric suite for object-oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, 1994.
[6] T. Miyake, Y. Higo, and K. Inoue, "A software metric for identifying extract method candidates," *IEICE Trans. Inf. & Syst.(Japanese Edition)*, vol. J92-D, no. 7, pp. 1071–1073, 2009.
[7] B. Henderson-Sellers, *Object-oriented metrics : measures of complexity*. Prentice-Hall, 1996.
[8] X. Wang, L. Pollock, and K. Vijay-Shanker, "Automatic segmentation of method code into meaningful blocks to improve readability," in *Proc. WCRE 2011*, Oct. 2011, pp. 35 –44.
[9] L. M. Ott and J. M. Bieman, "Program slices as an abstraction for cohesion measurement," *Information and Software Technology*, vol. 40, no. 11-12, pp. 691 – 699, 1998.

[10] J. Bieman and L. Ott, “Measuring functional cohesion,” *IEEE Trans. Softw. Eng.*, vol. 20, pp. 644–657, 1994.

[11] L. Ott and J. Thuss, “Slice based metrics for estimating cohesion,” in *Proc. METRICS 1993*, May 1993, pp. 71–81.

[12] M. Weiser, “Program slicing,” in *Proc. ICSE 1981*, Mar. 1981, pp. 439–449.

[13] S. Counsell, S. Swift, A. Tucker, and E. Mendes, “Object-oriented cohesion as a surrogate of software comprehension: an empirical study,” in *Proc. SCAM 2005*, Sept. 2005, pp. 161 – 169.

[14] N. Gold and K. Bennett, “Hypothesis-based concept assignment in software maintenance,” *IEE Proceedings - Software*, vol. 149, no. 4, pp. 103 – 110, Aug. 2002.