A Flexible Method for Segmentation in Concept Assignment

Nicolas Gold and Keith Bennett
Research Institute in Software Evolution.
Department of Computer Science,
University of Durham,
Durham, DH1 3LE, UK.
{N.E.Gold, Keith.Bennett}@durham.ac.uk

Keywords: Program Comprehension, Self-Organising Maps, Concept Assignment, Software Tools

Abstract

Software comprehension is one of the most expensive activities in software maintenance and many tools have been developed to help the maintainer reduce the time and cost of the task. Of the numerous tools and methods available, one group has received relatively little attention: those using plausible reasoning to address the concept assignment problem. This problem is defined as the process of assigning descriptive terms to their implementation in source code, the terms being nominated by a maintainer and usually relating to computational intent. *It has two major research issues:*

- Segmentation*: finding the location and extent of concepts in the source code.*
- Concept Binding*: determining which concepts are implemented at these locations.*

We present a new concept assignment method for COBOL II: Hypothesis-Based Concept Assignment (HB-CA). It employs a simple knowledge base to model concepts, source code indicators, and inter-concept relationships. The library and source code are used to generate hypotheses on which segmentation and concept binding are performed. An implementation of a prototype tool is described, and the results from a comprehensive evaluation using COBOL II sources summarised.

1. Introduction

Software maintenance is an important part of the software lifecycle, typically accounting for at least 50 percent of the total lifetime cost of a software system [10]. Consequently, it is desirable to reduce the cost of software maintenance whilst preserving the quality of the software system, maintenance process, and maintainer's understanding.

Many authors have acknowledged the central role and high cost of software comprehension within software maintenance, either directly (e.g. [12], [16]), or indirectly, as a consequence of software complexity (e.g. [1]). A common approach to reducing the cost of the maintenance process is the provision of automated assistance to software maintainers. Expert maintainers organise their knowledge around algorithms and functional characteristics within their domain of expertise [11]. The work presented here is aimed at assisting expert maintainers with software comprehension. Tilley and Smith claim that maintainers most lack tools that automatically identify algorithms, abstractions, and domain concepts in software [17]. Evidence that higher-level semantic knowledge reduces maintenance effort [13] strengthens their case.

2. Concept Assignment

To meet the need for tools that identify algorithms, abstractions, and domain concepts in programs, the method described in this paper addresses the *concept assignment problem*. The term was introduced by Biggerstaff et al. to describe the problem of assigning terms regarding computational intent to appropriate regions of source code [3]. The emphasis of the work presented here is on *automatic* concept assignment with minimal user involvement, although the activity can also be performed semi-automatically or manually. The

emphasis is particularly on *plausible reasoning concept assignment systems* as these tend to have linear computational growth with the length of the source code under analysis [3].

There are two examples of existing plausible reasoning concept assignment systems: DM-TAO (part of the DESIRE toolset) [3], and IRENE [6]. These systems adopt different approaches; DM-TAO has a complex knowledge base and inference engine driven by a connectionist network, IRENE uses rule-based concept acquisition techniques to retrieve business knowledge from COBOL programs.

Two major research issues can be identified within the overall concept assignment problem:

- *Segmentation*: finding the location and extent of concepts in the source code.
- *Concept Binding*: determining which concepts are implemented at these locations.

Segmenting a program involves grouping pieces of conceptual information generated from the source code. Concept binding involves analysing these groups for the most plausible concept assignment for each.

3. Hypothesis-Based Concept Assignment

The Hypothesis-Based Concept Assignment (HB-CA) method is a three-part non-interactive process. It operates on the procedure division of IBM COBOL II programs (although a complete program is provided as input). The decision to address only the procedure division was taken to reduce the scope of the research problem initially. We acknowledge that much useful information can be derived from the data division and future work may incorporate such analysis.

The three stages of HB-CA are: Hypothesis Generation, Segmentation, and Concept Binding.

The flow of control and data is sequential. The process begins with hypothesis generation from source code. This is followed by segmentation of the hypotheses to determine regions of conceptual focus in the program. Finally, concept binding finds the dominant concept in each segment.

Each stage uses a knowledge base termed the library.

3.1 Knowledge Base

It is anticipated that the maintainer, or some other person responsible for knowledge base maintenance, will construct the library, possibly using automated assistance such as that described in [15]. This would take place before the first use of HB-CA and the knowledge base content then could be improved as the maintainer gains experience.

There are two entities in the library: concepts, and indicators. Concepts are the terms nominated by the maintainer to describe items or activities in the domain. Indicators are evidence for concepts expressed in the implementation language, in this case IBM COBOL II.

The library encodes two types of relationship: Indicator-Concept, and Concept-Concept.

The indicator-concept relationship maps evidence for a concept to that concept. Concept-concept relationships map concepts to others to form composites and specialisations.

Indicators have a number of attributes: Name, Class, and Data. The *name* is a string used within the library to identify the indicator and provide an abstraction from the actual data. The *class* refers to the type of feature represented. There are four classes: identifier, keyword, comment, and segment boundary. This allows the indicator recognition process to filter indicators in the library for those appropriate to the search method being employed. Segment boundaries are not represented explicitly in the library but are generated from the subroutine structure of the code being analysed. The *data* is the actual evidence to be found in the source code. Alternatively, it may be a reference to another container for the data. The latter would be appropriate for complex indicators such as code fragments.

Concepts have three attributes: Name, Type, and Level. The *name* is a string to identify the concept, i.e. the nominated descriptive term. The *type* is either *action* or *object*. Action concepts are those that do something (typically, the name of an action concept is a verb, e.g. Read). Object concepts are those things on which action concepts operate (typically, the name is a noun, e.g. File). The classification allows greater control of the concept binding search than if none were used. Additionally, in combination with the relationships described below, it can help to reduce the size of the knowledge base required to represent complex concepts. Concept typing is used by various methods including

DM-TAO (see [3]). The *level* is either *primary* or *secondary*. Primary concepts represent the most general form of a particular concept; secondary concepts represent more specialised forms of primary concepts, e.g. File might be primary, MasterFile might be secondary. This information is required to help the method degrade its performance gracefully in the event of conflicting evidence. It allows the search methods to select a more general form of a concept if the evidence for specific versions is ambiguous.

The indicator-concept relationship, termed *indicates*, is formed by joining indicators to the concepts for which they provide evidence.

There are two concept-concept relationships in the library: *composition*, and *specialisation*. Composition relationships are formed by joining primary action concepts to primary object concepts. This forms an action:object structure (essentially a verb and noun construction) to convey more information to the maintainer (e.g. Read:File rather than merely Read). Creating a composition of two primary concepts also produces a series of implied composites with all specialisations of the primary object concept. These are not stored in the library but are used as required by the segmentation and concept binding methods. Specialisation relationships are formed by linking secondary concepts (i.e. specialisations) to primary or other secondary concepts. Multiple inheritance is not permitted.

3.2 Hypothesis Generation

The hypothesis generation stage takes source code as its input. Using information contained in the knowledge base, it scans the source code for indicators of various concepts. When an instance is found and matched, a hypothesis for the appropriate concept is generated. Matching is performed using a variety of flexible criteria. The resulting collection of hypotheses is ordered by the position of the indicators in the source code.

3.3 Segmentation

The segmentation stage takes the sorted hypotheses and attempts to break them into segments. Initially, this is performed using hypotheses for primary segmentation points (COBOL II section boundaries). Each of the initial segments is analysed to determine whether it has the potential to contain a number of smaller segments. If this is the case, a self-organising map is used to establish areas of conceptual focus within the segment. These areas are analysed and smaller segments created if necessary. The output of the stage is a collection of segments, each containing a number of hypotheses. This stage is discussed in more detail below.

3.4 Concept Binding

Concept binding analyses each segment's hypotheses to determine which concept has the most evidence. It exploits relationships in the knowledge base to generate conclusions, and scores these on the basis of concept occurrence. A number of disambiguation rules can be applied to choose between equally strong concepts. When a concept has been selected, the segment is labelled with the name of that concept. After all segments have been analysed and labelled, the results form the overall output of the method.

4. Flexible Segmentation

HB-CA's approach to segmentation is one of the most interesting parts of the process and merits more detailed discussion.

4.1 The Segmentation Problem

Segmentation is the problem of determining the location and extent of concepts within a piece of source code, to form segments that then can be labelled. It is a difficult problem because the boundaries between concepts can be confused and fuzzy to the point where two concepts may interleave. It presents a more difficult problem to plausible reasoning understanders, such as HB-CA, where this kind of information is not used. Figure 1 shows an example fragment of source code with two clearly separated concepts.

```
MOVE 'EXAMPLE' TO PRINT-LL.
MOVE '13' TO PRINT-CC.
CALL 'PRINT' USING P-PRINTLINE.
MOVE POLICY-NUM TO OUT-PNUM.
MOVE SCHEME-REF TO OUT-SREF.
CALL 'WRITE' USING OUT-REC.
```

Figure 1: Example Code Fragment Showing Separated Concepts

The first three lines indicate a Print concept; the last three indicate Write. In this situation, it is clear where the boundary between concepts falls. Figure 2 shows the same code but with the boundaries slightly blurred.

```
MOVE ‘EXAMPLE’ TO PRINT-LL.
MOVE ‘13’ TO PRINT-CC.
MOVE POLICY-NUM TO OUT-PNUM.
CALL ‘PRINT’ USING P-PRINTLINE.
MOVE SCHEME-REF TO OUT-SREF.
CALL ‘WRITE’ USING OUT-REC.
```

Figure 2: Example Code Fragment Showing Slightly Merged Concepts

There are still two distinct areas of conceptual focus although the boundary between them is now fuzzy. The final version of this example, shown in Figure 3, demonstrates the concepts when completely merged.

```
MOVE ‘EXAMPLE’ TO PRINT-LL.
```

SOMs have a two layer topology with an input layer the same size as the number of components in the input vectors, and an output layer usually in the shape of a two dimensional grid. Each output node has the same number of vector components as input nodes [14]. Every input node is connected to every output node. The output node vectors are initialised with random numbers. Learning takes place through the repeated presentation of training data vectors. There may be hundreds to thousands of repetitions. When a training vector is presented, the Euclidean distance between the training vector and every reference vector stored in the output nodes is calculated. The output node that is closest to the training vector is declared the winner, and its reference vector is updated to reduce its Euclidean distance to the input. In addition, neighbouring nodes in the output layer are also moved proportionally closer to the input. After many repetitions, this process results in the spatial organisation of the input data in clusters of similar, neighbouring regions [14]. Over the course of training, the size of the neighbourhood and the amount by which Euclidean distances are updated (the learning rate) decrease to zero.

SOMs have many uses including natural language engineering [5], and the organisation of document collections [7].

4.4 SOMs for HB-CA

The SOM is useful in HB-CA because of its ability to cluster similar data items automatically. Spatial relationships in the segment's hypotheses can be preserved allowing nearby, similar concepts to be clustered together. Consequently, the fuzzy boundaries between areas of conceptual focus in the hypothesis list can be determined using the conceptual content of the list itself, rather than imposing an arbitrary division.

Employing a self-organising map within HB-CA entails solving some additional problems. First, the map must be automatically constructed and the data pre-processed into a vector form. Second, the trained map must be automatically interpreted; a task often left to the user in other SOM applications.

These are designed to ensure that a self-organising map will only be used if there is the potential to form clusters, i.e. the hypothesis list is big enough with a sufficient number of different concepts.

To use the segment's hypothesis list with a self-organising map, it must first be turned into a vector representation. A coding scheme must be devised whereby different concepts can be represented as vectors without implying any spatial relationship between them in a single dimension. It is not possible (or sensible) to represent Read as 1, Print as 2, and Update as 3 in the same dimension, since the ordering relation on integers does not hold for concepts. The solution to this problem arises from SOM work in natural language engineering and document classification. Honkela [5] suggests the use of binary vector components to represent categorical data such as the hypotheses in HB-CA. This is the approach that has been adopted.

Having established the data encoding, the map itself must be defined. The task of the SOM in HB-CA is to cluster hypotheses to enable *automatic* inspection of the output. Consequently, the number of output neurons should be no more than necessary. This creates a coarser granularity in the output space than might be used for visually inspected maps, but forces hypotheses into one of a few groups thus providing sufficient vector density at each neuron for it to be recognised as a cluster.

The method used to generate the number of output neurons is based on the assumption of a perfectly clustered input list, e.g.

Read, Read, Read, Print, Print, Print,
Update, Update, Update

With a minimum vector density per cluster of 3, the maximum number of achievable clusters is 3. If the list is less than perfectly clustered, the number of achieved clusters will be 3 or less since the best case (perfect clustering on input) cannot achieve more. Each output node in the map represents one cluster (once trained, it will trigger for several input vectors) and therefore in this example, the output layer would contain 3 nodes. A problem for this method can be illustrated by examining what might be considered a worst-case scenario. Assume an input list of the form:

Read, Write, Read, Write, Read, Write, Read, Write,
Read, Write

This data is ambiguous since it could be described as having no dominant concept (and hence no clustering). Alternatively, it could be split in half (two output nodes), the first half being dominated by Read and the second by Write. With still more subdivision possible it is hard to say how the data should be clustered, or to

determine a suitable size for the output layer using the analysis method suggested. This seems to be an intractable problem for this type of input but since such an even distribution of hypotheses is unlikely to occur often, the method based on perfect clustering is considered suitable for use in all cases.

Having established the number of nodes in the output layer, its shape must also be considered. The most common shape for SOM output layers is a rectangular grid with either a rectangular topology (where nodes update those above, below, left, and right) or a hexagonal topology (where nodes are regarded as having six sides and update those surrounding them accordingly). For the purpose of HB-CA, the output layer is defined as one-dimensional with a rectangular topology. This ensures that the mutual attraction of like hypotheses operates in one dimension only on the map. In theory, a larger two-dimensional map would also work well since the combination of sequence number and concept would ensure that nearby and similar hypotheses group at the same node. Using this type would introduce additional problems, e.g. deciding on the length of each side of the rectangle. This would be particularly difficult if the number of nodes could not be formatted in rectangular fashion

The formatted SOM now can be trained on the input vectors created from the segment's hypotheses. Training for HB-CA takes place in two stages as suggested in [8]. The first stage orders the reference vectors in the map. Training data is presented 1000 times. The second stage converges the reference vectors on their "correct" values with a smaller learning rate. Data is presented 10000 times.

When training is complete, HB-CA interprets the SOM by passing the input data through the map once more, taking note of which output node triggers for a particular input vector. Vectors are grouped by the node that they trigger (thus forming a cluster) and are translated back to a hypothesis representation. The particular node triggered by an input vector is not inherently important; it is the association of this input vector with others triggering the same output node that is significant.

The clusters must be analysed to ensure that the required minimum vector density, *min_vd*, is met. Every cluster with ≥ *min_vd* vectors (termed a valid cluster) is stored in a list. If every cluster is analysed and the list remains empty or has one element only, the segment is stored using its original boundaries (from segment boundary hypotheses). Processing begins again for the next segment.

If the list has more than one element, further analysis is required. It is possible that, although a number of valid clusters have been found, there are some hypotheses participating in clusters that do not meet the required density. HB-CA takes the approach of including this information in the valid clusters rather than ignoring it altogether. This ensures that all hypotheses considered

with scalability, i.e. the ability of HB-CA to work accurately on modules of code of any length.

Accurate concept assignment is important since mistakes could confuse the software maintainer, thus increasing, rather than decreasing, the cost of software comprehension. HB-CA should maintain its accuracy regardless of the length of program to which it is applied. In principle, if HB-CA can be accurate on a single segment, there is no reason why it should be inaccurate when there are several segments, as each is analysed separately. Concept assignment is regarded as *accurate* if a segment contains an implementation of the concept specified. Concept assignment is regarded as *strictly accurate* if the concept is dominant in the segment (i.e. the segment is mostly concerned with implementing the concept specified).

To verify the scalability of HB-CA, an investigation was undertaken using a prototype implementation called HB-CAS. This was mostly written in Delphi and runs

Further confirmation is gained by comparing SOM usage and accuracy directly, as shown in Figure 6.

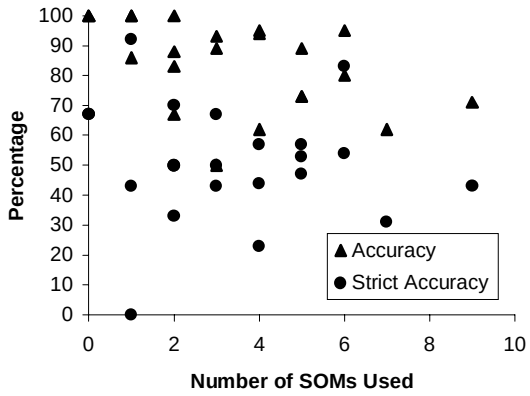

Figure 6: Graph to show the relationship between the Accuracy of Concept Assignment and Number of SOMs Used

The data indicates that the greater SOM usage arising from analysing larger programs correlates with a reduction in the accuracy of concept assignment. Hypothesis 1 would appear to be confirmed.

Recall that low concept assignment accuracy can be caused by poor quality segments. A further hypothesis is made to explain why greater SOM usage causes lower accuracy:

Hypothesis 2: SOM usage causes lower quality segments.

If the hypothesis is correct, it would explain the fall in accuracy with greater SOM usage

5.2 SOM-Related Segmentation Problems

There are two possible explanations for a link between greater SOM usage and lower quality segments:

1) The SOM is associating concepts that should not be clustered.
2) The algorithms that reallocate action-concept hypotheses from invalid clusters are introducing enough unrelated concepts to valid clusters to cause poor segment quality.

The most likely explanation can be determined by studying the balance between valid and invalid clusters at varying accuracies. If a low proportion of invalid clusters correlates with low accuracy, this would suggest that the SOM is causing the problem because the reallocation algorithms are not being used to a great extent. If there is a link between a high proportion of invalid clusters and low accuracy, this would indicate that the reallocation algorithms are at fault because they are being used often.

An investigation was undertaken on various programs that require SOM analysis. Sections that were subdivided by a SOM were examined to determine the number of valid and invalid clusters produced, and the accuracy of concept assignment for each resulting

to which they may have no conceptual affiliation, and adding entire invalid clusters to their neighbours without considering the content of either. When considering the problems the latter may cause, it is worth recalling that the SOM has associated the hypotheses in an invalid cluster, and consequently the neighbouring valid cluster gains a conceptually coherent group of hypotheses. Concept binding then could be hampered by both the general "noise" of unrelated individual hypotheses, or worse, it could be led in a completely different direction by conceptually coherent, but unrelated, groups of hypotheses. Increased SOM usage creates more opportunities for the reallocation algorithms to be employed. Hypothesis 2 is thus confirmed.

5.3 Possible Solutions

The reallocation algorithms would benefit from further research. One approach might be to use conceptual information from the hypotheses of invalid clusters, to bind them to conceptually similar neighbours. This might require some preliminary concept binding. Alternatively, the principle of preserving all of the original hypotheses could be rejected and invalid clusters ignored. Another idea might be to limit the number of hypotheses that can be added to a valid cluster, or limit the cluster size itself.

Another approach to improving the quality of segmentation might be to change the controlling parameters, *rec_thresh* and *min_vd*, which for these investigations were set to 1 and 3 respectively. Increasing *rec_thresh* would cause a reduction in the number of initial segments and hence concept assignments made (since more evidence would be required). Those segments that pass the threshold would be larger, having a reasonable amount of evidence. Smaller values of *rec_thresh* would allow more initial segments to be considered and increase the number of concept assignments. Given that smaller segments have been observed to produce more accurate concept assignment, smaller values of *rec_thresh* should produce more accurate results overall. The disadvantage of having smaller segments is that each hypothesis carries more weight (by representing a larger proportion of the body of evidence) than in larger segments. Consequently, a misleading indicator can cause greater problems. Individual hypotheses in larger segments have less influence on the overall concept assignment, so increasing *rec_thresh* may ensure that a reasonable body of evidence is considered, rather than just a few hypotheses.

Increasing *min_vd* would increase the number of invalid clusters by forcing valid clusters to contain more evidence. Decreasing *min_vd* may improve the quality of segmentation, but the resulting segments could be so small (since only one or two hypotheses for a concept would be required) that concept assignment would become pointless. There would no longer be a significant body of evidence to consider. A balance must be struck when setting the parameters, to make best use of the library on the source code being studied.

5.4 Average Performance

The overall performance of HB-CA is promising, achieving high mean and median accuracies as shown in Table 1.

	forced_specialisation = True	*forced_specialisation* = False
Mean Accuracy	84%, $\sigma = 14$	88%, $\sigma = 11$
Mean Strict Accuracy	56%, $\sigma = 19$	56%, $\sigma = 21$
Median Accuracy	89%	89%
Median Strict Accuracy	50%	56%

Table 1: Average Accuracy Values for HB-CA

6. Conclusions

We have presented a successful plausible-reasoning concept assignment method for COBOL II. It exhibits linear computational growth with the length of program under analysis. It can be applied to monolithic or poorly structured code, using the conceptual structure of the program to create segments for concept binding. The method shows a high degree of accuracy even with a simple knowledge base.

Despite theoretical claims that the accuracy should not decrease with longer programs, investigations indicate that such programs cause a wider variation in accuracy and a general drop in concept assignment performance. This is attributed to the greater use of SOMs when analysing larger programs, and the poorer quality of segmentation that can result.

Investigation of the cause of SOM-related segmentation problems revealed that the hypothesis reallocation algorithms are largely to blame for poor performance. This is not surprising given their naïve nature and

several strategies have been identified to address the problem.

Suggestions for further work on the method include the improvement of the reallocation algorithms, mapping the whole concept assignment problem to a SOM, increasing the richness of the knowledge base and/or the conceptual map, and the incorporation of data division analysis.

Another interesting line of research would be to use the HB-CA method to study the way segments and concepts change over several versions of the same program.

Acknowledgements

This work was funded by EPSRC as part of the Software As a Business Asset (SABA) project in the Systems Engineering for Business Process Change (SEBPC) programme. We also gratefully acknowledge the support of CSC and the Leverhulme Trust.

References

[1] **R.D. Banker, S.M. Datar, C.F. Kemerer, D. Zweig**, "Software Complexity and Maintenance Costs", *Communications of the ACM,* Vol. 36, No. 11, November 1993, pp. 81-94.

[2] **R. Beale, T. Jackson**, *Neural Computing: An Introduction*, IOP Publishing Ltd., 1992, ISBN 0852742622.

[3] **T.J. Biggerstaff, B.G. Mitbander, D.E. Webster**, "The Concept Assignment Problem in Program Understanding", *Proceedings of the Fifteenth International Conference on Software Engineering, Baltimore, Maryland, May 17-21, 1993,* IEEE Computer Society Press, 1993, pp. 482-498.

[4] **N.E.Gold**, "Hypothesis-Based Concept Assignment to Support Software Maintenance", *Ph.D. Thesis*, Department of Computer Science, University of Durham, November 2000.

[5] **T. Honkela**, "Self-Organising Maps in Natural Language Processing", *Ph.D. Thesis*, Helsinki University of Technology, 1997.

[6] **V. Karakostas**, "Intelligent Search and Acquisition of Business Knowledge from Programs", *Software Maintenance: Research and Practice*, Vol. 4, 1992, pp. 1-17.

[7] **S. Kaski, T. Honkela, K. Lagus, T. Kohonen**, "Creating an Order in Digital Libraries with Self-Organising Maps", *Proceedings of WCNN'96, World Congress on Neural Networks, San Diego, California, September 15-18, 1996,* Lawrence Erlbaum and INNS Press, Mahwah, NJ, 1996, pp. 814-817.

[8] **T. Kohonen, J. Hynninen, J. Kangas, J. Laaksonen**, "SOM_PAK: The Self-Organizing Map Program Package", *Technical Report A31, Laboratory of Computer & Information Science, Helsinki University of Technology,* ISBN 9512229471, January 1996.

[9] **T. Kohonen**, *Self-Organizing Maps, Second Edition*, Springer, 1997, ISBN 3540620176.

[10] **B.P. Lientz, E.B. Swanson**, *Software Maintenance Management*, Addison-Wesley Publishing Company, 1980, ISBN 0201042053.

[11] **A. von Mayrhauser, A.M. Vans**, "Program Comprehension During Software Maintenance and Evolution", *IEEE Computer,* Vol. 28, No. 8, August 1995, pp. 44-55.

[12] **A. von Mayrhauser, A.M. Vans, A.E. Howe**, "Program Understanding Behaviour During Enhancement of Large-scale Software", *Software Maintenance: Research and Practice*, Vol. 9, No. 5, 1997, pp. 299-327.

[13] **S. Ramanujan**, "An Experimental Investigation of the Impact of Individual, Program and Organisational Characteristics on Software Maintenance Effort", *Proceedings of the Second Americas Conference on Information Systems, Phoenix, Arizona,* J.M. Carey