Hypothesis-Based Concept Assignment in
Software Maintenance

Nicolas Gold
Department of Computation
UMIST
PO Box 88, Sackville Street
Manchester, M60 1QD, UK.
N.E.Gold@co.umist.ac.uk

Keith Bennett
Research Institute in Software Evolution.
Department of Computer Science,
University of Durham,
Durham, DH1 3LE, UK.
Keith.Bennett@durham.ac.uk
Abstract

Software maintenance accounts for a significant proportion of the lifetime cost of a software system. Software comprehension is required in many parts of the maintenance process and is one of the most expensive activities. Many tools have been developed to help the maintainer reduce the time and cost of this task but of the numerous tools and methods available, one group has received relatively little attention: those using plausible reasoning to address the concept assignment problem.

We present a new concept assignment method for COBOL II: Hypothesis-Based Concept Assignment (HB-CA). An implementation of a prototype tool is described, and the results from a comprehensive evaluation using commercial COBOL II sources summarised. In particular, we identify areas of a standard maintenance process where such methods would be appropriate, and discuss the potential cost savings that may result.

1. Introduction

Software maintenance typically accounts for at least 50 percent of the total lifetime cost of a software system [1]. It is desirable to reduce the cost of this activity whilst preserving the quality of the software system, maintenance process, and maintainer’s understanding.

Novice and expert maintainers understand code differently. Novices adopt a syntactic orientation, organising their knowledge structures around the program syntax, whereas experts organise their knowledge around algorithms and functional characteristics within their domain of expertise [2]. The comprehension model used initially depends
on the maintainer’s level of domain knowledge and code familiarity [3]. Models of program comprehension can be divided into three groups: top-down, bottom-up, and integrated.

Top-down understanding is typically applied when the code is familiar [2] and a good example of a top-down model is that defined by Soloway, Adelson, and Ehrlich [4], [5]. This model views the process of comprehension as the construction of a hierarchy containing goals. These goals are decomposed into structures called plans, which can describe a strategy for achieving a goal, a language independent problem solution, or be a code fragment implementing such a solution. Plans can be decomposed further into lower-level plans [2]. Brooks presents another top-down approach using a hierarchy of hypotheses [6].

Bottom-up models (typically used when the code is unfamiliar) suggest that the maintainer starts building a mental representation from the source code and chunks together elements into higher order structures. Chunking refers to the process of attaching descriptive labels to knowledge structures at various levels. Chunks can contain lower-level chunks with a description of how they interrelate [2]. Pennington’s model is an example of a bottom-up approach [7].

The integrated approach includes elements of the other two types by providing a framework within which both can be used as necessary. This model was suggested by von Mayrhauser and Vans and a number of studies have been performed to validate it [2] (see also [8, 9]).
All of the comprehension models have aspects that appeal to the personal experience of software maintainers. By including elements from top-down and bottom-up methods, the integrated meta-model of von Mayrhofer and Vans appears to be more comprehensive than the others. Glimpses of the meta-model can be seen in the other theories, e.g. to explain the experiences of professional maintainers, Brooks suggests that bottom-up understanding is a degenerate case of top-down understanding [6]. A more plausible explanation for this would seem to lie in the meta-model approach.

Von Mayrhofer and Vans identify three major components common to all models of comprehension: a knowledge base, a mental model, and methods for acquiring knowledge [2]. The knowledge base contains the maintainer’s general knowledge of the application domain, software engineering and maintenance knowledge, their experience and skills, and any other knowledge relevant to the task. The mental model is the internal, working representation of the software under consideration [2]; in other words, it contains the current state of comprehension. The methods for acquiring knowledge (and thus updating the knowledge base and mental model) vary from theory to theory. Littman et al. identify two major strategies: systematic, and as-needed. The systematic approach involves detailed line-by-line study of the program code whereas the as-needed strategy suggests localising the section of program required for a change before understanding it in greater depth [10].

Many authors have acknowledged the central role and high cost of software comprehension within software maintenance, either directly (e.g. [9], [11]), or indirectly, as a consequence of software complexity (e.g. [12]). A common approach to reducing this cost is the provision of automated assistance to software maintainers.
Tilley and Smith claim that maintainers most lack tools that automatically identify algorithms, abstractions, and domain concepts in software [13], and Ramanujan reports evidence that higher-level semantic knowledge reduces maintenance effort [14]. By providing an automatic assignment of descriptive terms to regions of source code, this sort of information is provided to the maintainer and it is reasonable to expect that they would not be required to expend as much effort in gaining an understanding of a particular source program.

In order to assist a software maintainer in understanding programs, we present a method to address the concept assignment problem: 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 [15].

The method we present, Hypothesis-Based Concept Assignment (HB-CA), is computationally inexpensive having linear computational growth in the length of the source under analysis. It performs concept assignment primarily using informal information such as identifier names and comments, and contains a novel approach to analysing monolithic and poorly structured code using a self-organising neural network. This associates similar and nearby concept hypotheses rather than relying entirely on their syntactic position.

2. Process Phases

The IEEE software maintenance standard defines seven stages for the maintenance process [16]:
a) Problem/modification identification, classification, and prioritisation;
b) Analysis;
c) Design;
d) Implementation;
e) Regression/system testing;
f) Acceptance testing;
g) Delivery.

The analysis, design, and implementation stages have particular relevance to the work presented here and the IEEE definition (see [16]) is discussed below to identify where software understanding is required.

2.1 Analysis
Analysis is an iterative process that has at least two components: feasibility analysis, and detailed analysis. The modification request, system and project documentation, and repository information are used to determine the feasibility and scope of the modification. Where documentation is inadequate and source code is the only reliable reference for the current system, reverse engineering is recommended. Various activities are required during analysis including the identification of elements involved in the modification, determination of the modification’s impact, identification of short and long-term costs, and implementation planning. The standard suggests that a preliminary modification list of those elements affected is created, e.g. software, specifications, database, and documentation. This involves some degree of software comprehension, probably at the system rather than program level, to determine which elements may be affected. It is interesting to note that although analysis requires the
identification of the elements involved, identifying the specific software modules affected is left until the design stage. This could make cost estimation extremely difficult in some circumstances.

Identifying the impact of the modification and building the preliminary modification list may involve ripple analysis. Ripple analysis involves assessing the effect of a change on other parts of a system and can be undertaken in various ways. The analysis stage of the software maintenance process is likely to require ripple analysis at the business-rule level primarily, as the identification of affected software modules is not addressed until the design stage.

Business rules have been defined as:

“A requirement on the condition or manipulation of data expressed in terms of the business enterprise or application domain.” [17]

A key idea is that the rule is stated at the level of the application domain, not of programming. Consequently, business rules are related closely to domain models but reflect the desires of a particular company, not the general features of a domain [17]. Business rules might be found in the formulae and conditions that define the growth and charging structure of a financial product such as a pension policy. These make certain requirements of manipulations on the entities involved in the management of the policy.
Ripple analysis in terms of business rules poses the following question: if one rule is changed, are others also affected? Finding affected rules may require the examination of documentation and software, with the cost of undertaking such analysis likely to be crudely proportional to the number of artefacts that need to be inspected. Business-rule ripple analysis can be seen as an example of the higher-order impact analysis that Tilley and Smith describe in [18]. Higher-order impact analysis allows the software engineer to analyse proposed changes at the application-domain level rather than the implementation-domain level [18].

The analysis phase produces a report that forms part of the input to the design phase.

2.2 Design

This phase uses the system and project documentation, source code, comments, databases, and the output of the analysis stage to design the modification to the system. The process includes identifying affected software modules, modifying their documentation, creating and identifying test cases, and updating the modification list. The whole phase involves software comprehension but two activities particularly require it: code ripple analysis, and module selection. Both are part of the process of identifying affected software modules.

Code ripple analysis answers a similar question to that posed for business rules above. In this situation however, the ripples are examined on the basis of potential changes to source code. This can be undertaken at a syntactic and semantic level, e.g. using a forward program slice, see [19]. Alternatively, it could be conducted conceptually in a similar manner to business-rule ripple analysis, with the difference lying in the type of
concept being considered. Code ripple analysis is more likely to be dealing with software engineering concepts than application-domain related concepts.

The cost of code ripple analysis is addressed extensively in the literature, usually in terms of specific algorithms. As the work presented in this paper is not concerned with particular methods for the process, we regard the cost as roughly proportional to the number and size of the artefacts examined.

Module selection is the process of determining which modules are affected by a proposed change. It can take place before and/or after ripple analysis and involves searching the code repository for instances of concepts or code that are known to require change. The cost can be seen as a function of the size of the code repository (in terms of total lines of source code) and the search method. Translating the behavioural description of a modification to its implemented counterpart can be extremely difficult. Concept-based search could assist with selecting the modules that need changing.

### 2.3 Implementation

Implementation involves making the specified changes to the system. The IEEE standard suggests that implementation should be commenced during the design phase, particularly if the change is complex, in order to better understand the modification. The standard defines four sub-processes: coding and unit testing, integration, risk analysis, and test-readiness review. Software comprehension is particularly required in coding and unit testing. Although the standard does not elaborate further on the coding sub-process, it is possible to break it into two parts: software module comprehension, and change implementation [20]. These parts may be iterative. Software module
comprehension involves studying the software module to be changed in order to understand where and how the change should be made. Once the module is understood, the change can be made. Achieving such understanding is a non-trivial task accounting for a very high proportion of the total cost of software maintenance. The effort of understanding can be seen as proportional to the size of the module being considered, although this relationship may not necessarily be linear since the maintainer may change comprehension strategy for different sizes of program (see [10]). Other factors such as the program complexity, quality of coding, and maintainer’s experience may also have an impact. Familiar modules are likely to take less time to comprehend than those not previously addressed.

2.4 Summary

The stages of the IEEE standard where there is a strong requirement of software comprehension have been discussed and cost factors identified in each. Four particular activities have been highlighted: business-rule ripple analysis, code ripple analysis, module selection, and software module comprehension.

3. Concept Assignment

To meet software maintainers’ needs 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 [21]. 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. In particular, we are concerned with plausible
*reasoning* concept assignment systems as these tend to have linear computational growth with the length of the source code under analysis [21]. In addition to the work presented here, there are two other examples of plausible reasoning concept assignment systems: DM-TAO (part of the DESIRE toolset) [21, 22], and IRENE [23]. 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.

The three methods share some characteristics, such as a knowledge base to represent concepts, but differ in several important ways. DM-TAO has a very rich concept representation but at the expense of complexity in the knowledge base. This is likely to incur high knowledge-base creation and maintenance costs and thus slowing the process of tool application. Unlike HB-CA and IRENE, knowledge-base updates can be undertaken semi-automatically. The lower complexity of the HB-CA and IRENE knowledge bases means that the setup and maintenance costs are lower but they do not have such a refined reasoning ability. HB-CA is a fully automated technique requiring no intervention once the knowledge base has been defined. Both IRENE and DM-TAO are interactive to allow the maintainer to guide their search for concepts. Both approaches have their advantages; the interactive approach may produce better-focussed results whereas the automatic approach can be included in a supporting tool environment.

In summary, HB-CA offers a simpler knowledge base than DM-TAO and it is likely that the creation and maintenance cost of this will be lower than either of the other two systems. It can also be incorporated within other software environments due to its
automatic nature. A more detailed comparison of the three approaches can be found in [15].

3.1 Research Issues
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.

4. Hypothesis-Based Concept Assignment
The Hypothesis-Based Concept Assignment method (see [15]) 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 procedure division provides enough information to obtain useful results although the data division may be included in future work. COBOL II was selected as the language for analysis owing to its wide use in commercial systems and the availability of real-world source code. The HB-CA method is not tailored specifically to this language (once the first stage is complete, all processing is language independent) but is likely to perform better on languages whose programs are of a similar style to COBOL II (i.e. imperative, non-object-oriented). HB-
CA does not account for variable scope or other language semantics except subroutine boundaries.

The method has three stages: Hypothesis Generation, Segmentation, and Concept Binding. Each stage accepts the output of the previous one as its input. Hypothesis generation accepts source code and produces an ordered list of hypotheses (a hypothesis represents a possible instance of a concept at a point in the code). Segmentation groups these hypotheses using their conceptual affiliation to produce a list of hypothesis segments. Finally, concept binding evaluates the evidence in the segments and assigns a concept to each. The overall output is a list of concepts, associated with regions of source code.

Each stage uses a knowledge base termed the library.

4.1 Knowledge Base (Library)

It is anticipated that the maintainer, or some other person with domain and software engineering knowledge will construct the library initially. As the method and library are used, the knowledge base content can be modified and improved as it has a relatively simple structure.

The library contains two entities: 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 (e.g. APS -> APSRecord). It is used to generate hypotheses. Concept-concept relationships map concepts to others to form composites (e.g. Read|File) and specialisations (e.g. File -> MasterFile). These are used to provide flexible disambiguation of conflicting results. Both types of relationship are sufficient to encode many concepts yet ensure that the knowledge base does not become too complex.

4.2 Stage 1: Hypothesis Generation

The hypothesis generation stage takes source code as its input. Using information contained in the knowledge base, it analyses the source code for indicators of various concepts. When an instance is found and matched (by comparing strings in the library to strings in the source code), a hypothesis for the appropriate concept is generated. Matching can be performed with some flexibility e.g. using sub-strings, synonyms, and case sensitivity. The resulting collection of hypotheses is ordered by the position of the indicators in the source code. The number of hypotheses generated varies greatly with the content of the knowledge base and source code being considered.

4.3 Stage 2: Segmentation

The segmentation stage takes the sorted hypotheses and attempts to group 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 (see [15] and [24]). The self-organising map offers the ability to form clusters without
knowing how many are required in advance. It is possible that other techniques may also offer this benefit but these have not been investigated as part of this work.

Self-organising maps (SOMs) (see [25]) are a form of neural network employing unsupervised, competitive learning. They perform a topological mapping of high-dimensional input data to a low-dimensional output space. If SOM analysis is required, the hypotheses generated in the first stage of HB-CA are translated into a high dimensional vector format and clustered on the output nodes of a SOM.

These clusters are analysed and, if sufficiently large (valid), then form the basis for new, smaller segments. Clusters that are not large enough (invalid) are merged (by moving the boundaries of the surrounding segments) with those that are valid. Cluster validity is determined by the presence of \( n \) or more action hypotheses in the cluster (where \( n \) is a user-defined threshold value). The output of the stage is a collection of segments, each containing a number of hypotheses.

### 4.4 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 using the frequency of concept occurrence in the segment. 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. The concept scoring method is biased towards generating composite concepts above all else (to provide maximal information). The
disambiguation rules exist to choose concepts where the result is not clear and it would be possible to suggest an equally valid set of rules with different priorities.

5. Evaluation

HB-CA has been evaluated extensively on a variety of criteria (see [15]) relating to both the characteristics of the method itself, and its application in the maintenance process. The method characteristics with particular relevance to the work described here are those concerned with practical applications of the method: scalability, and computational cost. Scalability is the ability of HB-CA to work accurately on modules of code of any length, and it is important that the computational cost is in reasonable proportion to that length, given the wide variation in size of commercial programs. The results are summarised here; a more detailed treatment can be found in [15] and [24].

5.1 Scalability Evaluation

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 running on Microsoft Windows. The self-organising map processing was implemented using the SOM_PAK provided by Kohonen’s group (see [26]) harnessed to a Delphi library. Each part of the system used INI files for input and output.

22 commercial COBOL II programs of various lengths (from about 100 to 1500 lines) were used to evaluate the method. The code comes from a financial services domain, is written in accordance with company standards, is heavily maintained, and has been in production for several years. The library used in these investigations contained 23 concepts, each having between 1 and 3 indicators. Most concepts related to software engineering although some domain aspects were included.

Initial attempts to measure the accuracy of the results used precision and recall metrics. However, it was difficult to gain a clear picture of the system’s performance with these since the reference data (in terms of scope and concept binding) did not always match up with the results, even when an inspection of the results indicated that they were correct. This is reflection of the problem that two human maintainers might have when looking at the same piece of code: they may interpret it differently. Thus, the human marking up the reference data may place one interpretation on the scope and binding of a concept, and the system places another (equally correct) interpretation in the same area of code. The solution to this problem (at least within the scope of this investigation) was to assess accuracy and strict accuracy in terms of the bindings and scopings made by the system. The judgement (made by the authors) was therefore: is this a reasonable interpretation of this section of code? The definitions of accuracy and
strict accuracy give some guidance about this. Clearly, a more extensive investigation involving professional software maintainers using the system in their workplace would be ideal but was beyond the scope of this work.

The results (see Figure 1 and Figure 2) showed that accuracy is maintained for programs of length less than 1000 lines. However, at around 1000 lines accuracy falls and starts to vary more widely, strict accuracy following a similar trend. This indicates that the expected scalability of HB-CA has not been achieved. We now discuss the reasons for this unexpected result.

Accurate concept binding relies on a good quality segment, i.e. a set of hypotheses that clearly indicate one concept. It follows that the lower a segment’s quality, the less likely the concept binding method is to accurately assign a concept. Further investigations were carried out to determine why larger programs cause lower accuracy and we found that such programs required more SOM analysis than smaller examples.

Since larger programs require greater use of SOMs, it is possible that the latter is the cause of lower accuracy. There is certainly a correlation when comparing SOM usage and accuracy directly (see [15] and [24]).

The next part of the investigation looked at why the SOM should cause low quality segments, and thus low concept assignment accuracy. Two explanations for a link between greater SOM usage and lower quality segments were considered:

1. The SOM is associating concepts that should not be clustered.
2. The algorithms that reallocate 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. The more invalid clusters that occur in a segment, the more often the reallocation algorithms are required. If a low proportion of invalid clusters correlates with low accuracy, it 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 [24].

Various programs (drawn from the same set) requiring SOM analysis were tested. Those sections that were analysed by a SOM were examined to determine the number of valid and invalid clusters produced, and the accuracy of concept assignment for each resulting segment [24].

The results (see [24]) indicated that higher proportions of invalid clusters lead to lower strict accuracy, although this is not reflected to the same extent in non-strict accuracy.

When considering non-strict accuracy, good results can be achieved with poorer segmentation because non-strict accuracy relies only on the presence of a concept within a segment. The stringent requirements of strict accuracy mean that “loose”
segmentation (where a large part of the segment is irrelevant to the concept) is more
evident in the results.

The conclusion that can be drawn is that the problems lie in the reallocation algorithms.
This is not surprising since the “equal-division” method of assigning invalid clusters
and hypotheses to their surrounding valid clusters is naïve. It causes “loose”
segmentation (and thus poor segment quality) by including hypotheses in segments to
which they may have no conceptual affiliation, and adding entire invalid clusters to their
neighbours without considering the content of either. Increased SOM usage creates
more opportunities for the reallocation algorithms to be employed [24].

Despite these difficulties, HB-CA achieved high mean and median accuracies on the
test data as shown in Table 1.

5.2 Computational Cost Evaluation
It is important to consider the impact of the source code being analysed on the
computational cost of HB-CA. This is because it is likely to change more frequently
than any other entity involved in the method. Plausible-reasoning concept-assignment
systems tend to have linear computational growth with the length of source code under
analysis [21]. Since HB-CA is of this type, it is reasonable to expect it to exhibit the
same computational growth characteristic.

An investigation was undertaken to verify the relationship. The execution time of a
module or part-module of HB-CAS was regarded as directly proportional to the
computational cost of the method it implements. The modules of HB-CAS supply these
timings, accurate to within 1 second. 20 source programs were selected from a set of 150. The selection criteria were to include the shortest and longest available programs, space the program lengths by approximately 50 lines, and ensure that programs were drawn from the same system (to ensure the applicability of the library for that domain).

The results of the investigation (see Figure 3) showed a clear linear relationship between the overall execution time and the length of program being analysed. More detailed analysis (see [15]) to compare the relative costs of each stage showed that segmentation and concept binding did not show a linear relationship so clearly, but hypothesis generation (which is linear) dominates the other costs to such an extent that the overall result does not make these smaller variation apparent. [15] contains a considerably more detailed exposition of both computational and spatial cost investigations.

6. Applications of HB-CA in the Maintenance Process

This section evaluates the potential applicability of HB-CA to various software maintenance tasks.

6.1 Business-Rule Ripple Analysis

Using HB-CA to assist with ripple analysis would require some additions to the method or its implementation to enable it to analyse multiple source files. This would be a wrapper supplying each candidate file to HB-CA and analysing the result of concept assignment. Business rules would be modelled in the library, and the modified method would be supplied with the rule being proposed for change. The library would need to be populated entirely with business rule concepts to avoid confusion with the
programming domain. Concepts found in programs that implement the “proposed change” rule would be presented as candidates for side effects of the change. The maintainer could then examine them and accept or reject these suggestions for further analysis. Figure 4 shows this process.

Section 2.1 stated that the cost of business-rule ripple analysis is crudely proportional to the number of artefacts inspected. One advantage of using HB-CA in this activity would be that many programs could be scanned quickly for potential side effects in the business rules, reducing the number that the maintainer is required to examine by hand. This application of HB-CA may have limited success given the difficulty of representing constraint information in the library.

HB-CA could not totally replace the maintainer because it cannot determine dependencies between rules beyond that of co-occurrence in the same program. In this sense, it does not undertake traditional ripple analysis as it does not predict the effect of a change, but only makes suggestions for potential side effects. If it is likely that related rules do co-occur, HB-CA could substantially reduce the size of the task by limiting the number of code items that require inspection.

6.2 Code Ripple Analysis
This is similar to business-rule ripple analysis but is more likely to occur in the design phase of the maintenance standard as part of identifying affected software modules. Code ripple analysis is used to determine the effect of changes to the source code. There are various methods to perform this using syntactic and semantic techniques (e.g., forward program slicing, see [19]), but HB-CA could perform it on a conceptual level.
There is little difference between code ripple analysis and business-rule ripple analysis, except in the type of concept being considered. Business rules are closer to the application domain than the type of concepts that usually would be used for code ripple analysis. These would probably be nearer to the implementation domain. The process of using HB-CA for this activity would be much the same as that described in section 6.1, although the library would probably contain lower-level concepts in addition to those modelling business rules.

Section 2.2 stated that the cost of code ripple analysis is cruelly proportional to the number and size of the artefacts examined. Potential cost savings could result from the reduced size of the code repository requiring manual inspection, on the principle that co-occurrence of concepts indicates some dependency. As discussed earlier, relying solely on this relationship prevents HB-CA from fulfilling the requirements of traditional ripple analysis.

6.3 Module Selection

HB-CA can assist with this activity to a greater extent than it can with ripple analysis. Once the concepts to be changed are known, the task of finding instances of them in the code base can be extremely time consuming. Using a similar wrapper to that described earlier, the concept required can be supplied to HB-CA (as the only concept in the library) and programs that implement it can be found. These would be the modules requiring change.

Section 2.2 described the cost of module selection as a function of the size of the code repository and the search method. The cost savings from this application of HB-CA
could be quite considerable since the maintainer does not need to participate in the selection activity if the wrapper is used. If HB-CA is employed in its current form (i.e. analysing one module at a time), reduced cost could still be achieved because the maintainer would not need to read every program entirely. The concept list would show whether the concept to be changed exists in the code. Concept-based search could perform better than some other automated methods of examining source code (e.g. plan recognition) because it has linear computational growth with the length of source code being analysed.

6.4 Code Reuse

Although not explicitly placed in the standard process, code reuse can substantially reduce the cost of software maintenance. Using HB-CA in a similar manner to module selection could facilitate this activity. It might be particularly helpful with languages such as COBOL II that do not lend themselves to populating reuse libraries. The code repository could be searched for instances of a particular concept required for implementation in another program. HB-CA could be particularly helpful since SOM-based segmentation may be able to identify parts of subroutines that implement the required concept, even if the whole routine is not relevant.

6.5 Implementation Activities

One of the steps in the implementation stage of the standard process is coding and unit-testing. Without HB-CA in the understanding part of the coding sub-step, the maintainer must work from only one knowledge source, performing the labelling process themselves (either mentally, or explicitly using an external representation). Figure 5 shows this.
Employing HB-CA allows some *a priori* analysis of the code, giving the maintainer two knowledge sources from which to work, and relieving them of the labelling task (see Figure 6).

**7. Conclusions**

We have identified several stages of the IEEE standard maintenance process that require software comprehension and have presented a successful plausible-reasoning concept assignment method as a way of reducing the cost of maintenance. The method exhibits linear computational growth with the length of program under analysis and can be applied to monolithic, poorly-structured, and heavily-maintained code. This is due to its ability to use the conceptual structure of the program to create segments for concept binding. The method shows a high degree of accuracy with even a simple knowledge base.

Investigations of the method’s scalability found that longer 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. The cause of SOM-related segmentation problems appears to be the hypothesis reallocation algorithms. This is not surprising given their naïve nature. It may be possible to improve their performance by considering the content of the clusters produced, or by ignoring the invalid clusters entirely.
Finally, we have related the application of HB-CA to the activities and process stages identified at the beginning of the paper. This has shown how methods like HB-CA might reduce the cost of software maintenance by providing concept-based search methods on a code base, and providing the maintainer with an additional knowledge source from which to work.

Further work on the method may 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.

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 and undertaken at Durham University. We also gratefully acknowledge the support of CSC and the Leverhulme Trust.

References


    Large Scale Maintenance’. Proceedings of the Sixteenth International Conference
on Software Engineering, Sorrento, Italy, May 16-21, 1994, IEEE Computer

IEEE Transactions on Software Engineering, 1984, 10, (5), pp. 595-609

Comprehension of Computer Programs’, in ‘The Nature Of Expertise’ (M.T.H.
0805804048, pp. 129-152)


295-341

Software Adaptation Tasks’. Proceedings of the International Conference on
Software Maintenance, Bethesda, Maryland, November 16-19, 1998, IEEE

Behaviour During Enhancement of Large-scale Software’, Software Maintenance:
Research and Practice, 1997, 9(5), pp. 299-327

and Software Maintenance’, in ‘Empirical Studies of Programmers: First
Workshop, June 5-6, 1986, Washington, DC’ (E. Soloway, S. Iyengar (editors),
Ablex Publishing Corporation, Norwood, New Jersey, 1987 (second printing),
ISBN 0893914630)


Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, December 1996


Table 1: Average Accuracy Values for HB-CA [24]

<table>
<thead>
<tr>
<th></th>
<th>forced_specialisation = True</th>
<th>forced_specialisation = False</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean Accuracy</td>
<td>84%, $\sigma = 14$</td>
<td>88%, $\sigma = 11$</td>
</tr>
<tr>
<td>Mean Strict Accuracy</td>
<td>56%, $\sigma = 19$</td>
<td>56%, $\sigma = 21$</td>
</tr>
<tr>
<td>Median Accuracy</td>
<td>89%</td>
<td>89%</td>
</tr>
<tr>
<td>Median Strict Accuracy</td>
<td>50%</td>
<td>56%</td>
</tr>
</tbody>
</table>
Figure 1: Graph to show the relationship between the accuracy of concept assignment and program length (non-strict accuracy)

Figure 2: Graph to show the relationship between the accuracy of concept assignment and program length (strict accuracy)

Figure 3: Graph to show the relationship between program length and the total execution time

Figure 4: HB-CA Used for Ripple Analysis

Figure 5: Program Comprehension without HB-CA

Figure 6: Program Comprehension with HB-CA