Code Extraction Algorithms which Unify Slicing and Concept Assignment

Mark Harman[1] **Nicolas Gold**[2] **Rob Hierons**[1] **Dave Binkley**[3]

[1]Brunel University
Uxbridge, Middlesex
UB8 3PH, UK.

[2]UMIST
Manchester
M60 1QD, UK.

[3]Loyola College
Baltimore
MD 21210-2699, USA.

Keywords: slicing, concept assignment, source code extraction

Abstract

One approach to reverse engineering is to partially automate subcomponent extraction, improvement and subsequent recombination. Two previously proposed automated techniques for supporting this activity are slicing and concept assignment. However, neither is directly applicable in isolation; slicing criteria (sets of program variables) are simply too low level in many cases, while concept assignment typically fails to produce executable subcomponents.

This paper introduces a unification of slicing and concept assignment which exploits their combined advantages, while overcoming their individual weaknesses. Our 'concept slices' are extracted using high level criteria, while producing executable subprograms. The paper introduces three ways of combining slicing and concept assignment and algorithms for each. The application of the concept slicing algorithms is illustrated with a case study from a large financial organisation.

1 Introduction

For program comprehension and reverse engineering it is important to have automated techniques for extracting executable subcomponents according to high level extraction criteria. These components need to be semantically related to the original (so that they can be executed in isolation), while the criteria for selection may need to identify disparate sections of diverse code which will have to be married together. Therefore, the problem is to be able to automatically produce programs which answer questions of the form: Given an original program, construct the simplest program that, for example performs the same master file update operation or which closes down the reactor under the same conditions.

Program slicing and concept assignment are automated source code extraction techniques that take a criterion and program source code as input and yield parts of the program's source code as output. Therefore, they suggest themselves as natural candidate solutions to this problem. Slicing has the advantage that the extracted code it produces can be executed as a program in its own right, but the disadvantage that the criterion must be expressed at the low level of program variables. Concept assignment has the advantage that the extraction criterion is expressed at just the right level (in terms of concepts such as 'master file', 'error recovery' and 'log update'), but the disadvantage that the code fragments it extracts cannot be compiled and executed as a separate program. Thus, each technique overcomes the difficulty associated with the other.

This paper shows how slicing and concept assignment can be combined to produce better results than either is capable of individually. The contributions of this paper can be summarised as follows.

- A framework for combining Slicing and Concept Assignment is introduced
- Algorithms are introduced for
  - Executable Concept Slicing
  - Key Statement Analysis
  - Concept Dependency Analysis
- The application of the concept slicing approach to reverse engineering is illustrated with a case study

The rest of the paper is organised as follows. Section 2 briefly reviews slicing and concept assignment to make the paper self-contained. It can safely be skipped by a reader familiar with both techniques. Section 3 presents a framework for unifying slicing and concept assignment, suggesting three new techniques which combine slicing with concept assignment. Algorithms for these three techniques:

Executable Concept Slicing (ECS), Key Statement Analysis (KSA) and Concept Dependency Analysis (CDA) are introduced in sections 4, 5 and 6 respectively. Section 7 presents a case study involving a financial payment system, which illustrates the use of the concept slicing algorithms introduced in sections 4, 5 and 6. Section 8 concludes and Section 9 gives directions for future work.

2 Background

This section provides some background, definitions and notation for slicing and concept assignment which are used in the remainder of the paper.

2.1 Slicing

Program slicing [31] is defined with respect to a 'slicing criterion'. Slicing uses dependence analysis to isolate those parts of a program that potentially affect the slicing criterion.

Traditionally 'parts of the program to be isolated' have been restricted to statements and predicates and the slicing criterion has been defined in terms of a set of variables and a point at which their values are of interest. More recent work has extended traditional slicing by considering novel slicing criteria involving conditions and test adequacy properties [5, 15]. The techniques for isolation of statements have also broadened from statement deletion to allow for more general transformation [4, 12, 30].

This paper will be concerned solely with syntax-preserving static slicing, which will be used both to refine and to extend the results of concept assignment. In all cases, slices will be constructed for a set of nodes of a program's Control Flow Graph (CFG). This means that the slicing criterion will simply be a set of n statements $\{s_1, \ldots, s_n\}$.

Definition 1 (Slice)
A *slice* of a program p for the slicing criterion $\{s_1, \ldots, s_n\}$ is an executable subprogram, s, constructed from p by statement deletion, such that s behaves identically to p with respect to the sequence of values computed at each of the statements in $\{s_1, \ldots, s_n\}$. The slice of a program p w.r.t a set of statements S will be denoted $Slice(p, S)$.

This definition of a slice is essentially the executable version of the definition adopted by the System Dependence Graph approach of Horwitz et al. [16]. Typically, work on the System Dependence Graph (SDG) defines it to contain a set of 'final use' vertices for each variable. The SDG is so-constructed to guarantee the existence of such a vertex for each variable. This allows slices to be constructed for a variable in terms of its final use vertex.

Definition 2 (Final Use Vertex)
$FinalUse(p, v)$ is the final use vertex of variable v in program p.

The dependence graph itself can be useful in analysing the distance between a slice node and some other node in the slice.

Definition 3 (SDG Distance)
Given statements s and s' of a program p, the distance, $Dist(p, s, s')$ is the length of the *shortest* path between s and s' in the SDG of p. If there is no path from s and s' in the SDG of p, then $Dist(p, s, s')$ is undefined.

2.2 Concept Assignment

The concept assignment[1] problem is defined as "a process of recognising concepts within a computer program and building up an 'understanding' of the program by relating recognised concepts to portions of the program, its operational context and to one another [3]." It can be undertaken by intelligent agents (tools), with three distinct approaches being adopted [3]:

1. Highly domain specific, model driven, rule-based question answering systems that depend on a manually populated database describing the software system. This approach is typified by the Lassie system [7].

2. Plan driven, algorithmic program understanders or recognisers. Two examples of this type are the Programmer's Apprentice [28], and GRASPR [32].

3. Model driven, plausible reasoning systems. Examples of this type include DM-TAO [3], IRENE [17], and HB-CA [9, 10].

Biggerstaff et al. claim that systems using approaches 1 and 2 are good at completely deriving concepts within small-scale programs but cannot deal with large-scale programs due to overwhelming computational growth. Approach 3 systems can easily handle large-scale programs since their computational growth appears to be linear in the length of the program under analysis but they suffer from approximate and imprecise results [3].

We are concerned with plausible reasoning systems (category 3 above) and all references to concept assignment in this paper should be taken as referring to this kind of system. Plausible reasoning systems are of particular interest because they are scalable

[1] Note: concept assignment is a wholly *different* technology from formal concept analysis (FCA) (sometimes just called 'concept analysis').

and are theoretically capable of assigning higher-level concepts than some of the other approaches. The assignment is based on the evidence available in the code being analysed from which a 'best guess' is taken; reasoning is thus based on plausibility rather than deduction. In addition to the common application of concept assignment in helping maintainers to comprehend programs, Cimitile et al. [6] have suggested it as a way of validating the adequacy of a candidate criterion when identifying suitable modules for reuse.

Hypothesis-Based Concept Assignment (HB-CA) [9, 11] is one of the most recent examples of a plausible-reasoning concept assignment approach. It deals with the part of the concept assignment problem that involves relating recognised concepts to portions of a program. HB-CA uses a simple knowledge base to encode the relationships between concepts and potential evidence for them in source code. It is this approach that we propose to combine with program slicing. The following definition introduces the notation we will use to denote concept assignment.

Definition 4 (Concept)
A concept c, named n, of a program p is constructed with respect to a domain model D. The concept consists of a tagged contiguous sequence of code from p, for which there is evidence (according to D) that the sequence implements the concept named n. For a concept c, $Tag(c)$ refers to the name of the concept c, while $Statements(c)$ refers to its statements. For a program p and domain model D, $Concepts(p, D)$ refers to the set of all concepts assigned to p according to D.

Figure 1 shows a fragment of a domain model (which will be used in the case study in Section 7). In a domain model, concepts are classified into actions and objects and may be composed or specialised. Each concept has a number of indicators which represent the potential source code evidence for the concept. For every piece of source code evidence that is found, a hypothesis is generated for the appropriate concept. Segments (contiguous groups of hypotheses defining a contiguous region of source code) are formed from the resulting list of hypotheses using conceptual density and program syntax to define the boundaries. The dominant concept (i.e. the one for which there is most evidence) in each segment is assigned to the appropriate region of source code.

3 A Framework for Unifying Slicing and Concept Assignment

This section presents a framework of notation and requirements for developing a combined slicing and concepts approach. The term 'concept slice' will be used to refer to the result of any combination of slicing and concept assignment. Figure 2 depicts, in overview, the three types of concept slice considered in this paper.

3.1 Executable Concept Slicing (ECS)

Executable Concept Slicing (ECS) is the basic starting point for the approach we advocate. An ECS is formed using slicing to augment the results of concept assignment to make the concept an executable sub-program.

More formally, an algorithm for ECS is a function which takes a program, p, and a domain model, D, and produces a set of executable sub-components, one per concept in the program p according to D. Each returned concept must be executable and, when executed, the computation captures the computation on the associated concept of p w.r.t. D. In forming an ECS, the set of statements tagged with the concept name may no longer be contiguous.

The end of the segment identified in concept assignment will be treated as an end of program vertex. For example, in the case of COBOL, this can be achieved by inserting a `STOP RUN` statement at the end of the segment of code assigned to the concept.

The ECS will be further refined using key statement analysis, as described in the next section.

3.2 Key Statement Analysis (KSA)

Given a concept (and/or concept slice), some statements will be more important than others; they will contribute more to the computation embodied by the concept. The more important statements are regarded as the 'key' statements of the concept. Key Statement Analysis (KSA) is an analysis step which aims to determine the key statements in a concept. The approach can be applied to both concepts and to concept slices.

More formally, an algorithm for KSA is a function which takes a program and a concept assigned within it, and returns a function which describes the relative weight of each statement in the concept. The weight is represented as a function, form statements, STATEMENT, to real numbers, $\mathbb{R}$. If the function returned is f then $f(s)$, denotes the weight of statement s

A simple approach identifies a subset of the statements as being key. A more elaborate approach, assigns weights to each statement, indicating relative key*ness*. These weightings will be real numbers in the range 0 to 1 and so the simple case is merely a special case in which the only two outcomes are 0 and 1.

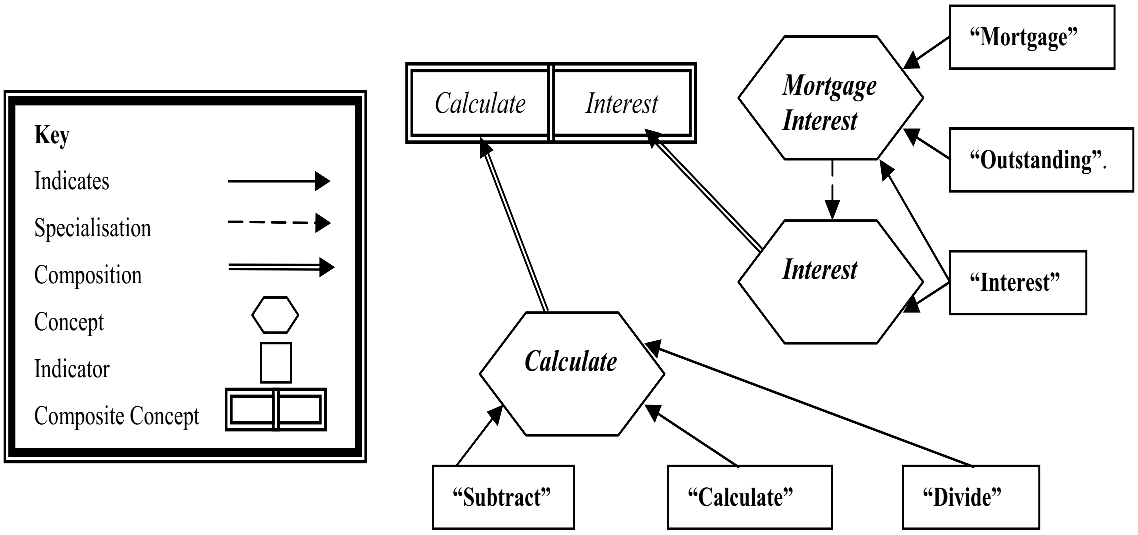


Figure 1: A Fragment of a Domain Model

3.3 Concept Dependency Analysis (CDA)

To perform concept assignment, an initial domain model is created by the software engineer (based on their experience). This model ought to be improved as the method is used. Unfortunately, using traditional concept assignment, there is no guidance to indicate which concepts occur together frequently nor to elucidate the inter-concept relationships which evolve as the analysis process iterates. Therefore, the model is improved only by serendipity and in a poorly defined ad-hoc manner. A clearly defined and tool-assisted feedback approach is required to support the disciplined and systematic evolution of the model, facilitating process improvement over successive analyses.

More formally, an algorithm for Concept Dependency Analysis (CDA) takes a program and a domain model and produces a concept dependence graph. A concept dependence graph is a directed graph, in which the nodes are concepts and the edges are weighted. Thus, formally, the concept dependency graph is a set of triples, such that triple (c, c', w) is in the graph iff there is an edge from concept c to c' with weight w. The concept graph is thus a weighted relation on the set of concepts.

3.4 Principal Variables

In order to form concept slices for KSA and CDA it will be necessary to determine the principal variables of an arbitrary set of statements. The principal variables are those which might be considered to be the result of the set of statements.

As Bieman and Ott point out in their work on slice-based cohesion measurement [18, 25], the decision as to what variables are 'principal' is somewhat arbitrary; changing it can, of course, alter the results of the algorithms upon which it is based. Therefore, the definition of what constitutes a principal variable should be treated as a parameter of the concept slicing approach advocated here. The definition below will be used as a working definition for the case study in Section 7, and is derived from Bieman and Ott.

Definition 5 (Principal Variable)
A variable v in a set of statements S is a principal variable iff it is either

- global and assigned in S
- call-by-reference and assigned in S
- the parameter to an output statement of S

Given a set of statements S, $PV(S)$ will be used to denote the set of principal variables of S.

4 ECS Algorithm

The ECS algorithm is presented in Figure 3. The algorithm is straightforward. The statements of the concept form the set of statements for the slicing criterion. Slicing on these statements adds to the concept, all statements of the original program required to ensure that the concept statements faithfully mimic (in the concept slice) their behaviour in the original program.

Name	Purpose	Type	Potential Applications
Executable Concept Slicing (ECS)	To form an executable sub-component	Inter-Concept Analysis	Reuse and re-engineering
Key Statement Analysis (KSA)	To refine a concept	Intra-Concept Analysis	Comprehension and reverse engineering
Concept Dependency Analysis (CDA)	To identify inter-concept relationships	Inter-Concept Analysis	Domain model improvement

Figure 2: Overview of the Concept Slicing Framework

```
function ECS(PROGRAM p, DOMAINMODEL D)
returns: set of PROGRAM

let {c_1, ..., c_n} = Concepts(p, D)
for each c_i ∈ {c_1, ..., c_n}
   let ECS_i = Slice(p, Statements(c_i))
   let Tag(ECS_i) = Tag(c_i)
endfor
return {ECS_1, ..., ECS_n}
```

Figure 3: The Executable Concept Slicing Algorithm

```
function KSA_BO(PROGRAM p, CONCEPTSLICE c)
returns: function from STATEMENT to {0, 1}

for each variable v_i in PV(c)
   let s_i = Slice(p, {FinalUse(Statements(c), v_i)})
endfor
let Tight = ⋂_i s_i
let KS = Statements(c) ∩ Tight
return λx. if x ∈ KS then 1 else 0
```

Figure 4: Key Statement Analysis 'BiemanOtt' Style

5 KSA Algorithm

A simple KSA algorithm is presented in Figure 4. The function KSA_{BO} takes a program and a concept assigned within it and returns a function which describes the relative weight of each statement in the concept. In this case, the returned result is either 0 or 1, with 1 signifying that the statement is a *key* statement and 0 signifying that it is not.

The idea is to use the set of principal variables in the concept to form a set of slices. The intersection of these slices contains the statements which contribute to the computation of every principal variable; in other words, the *key* statements of the concept.

We call this the 'BiemanOtt-style' algorithm, because it is inspired by Bieman and Ott's work on measuring cohesion using slicing [2, 18, 25, 26, 27]. Specifically, the intersection of slices on principal variables is the set used to compute the 'Tightness' metric introduced by Ott and Thuss [26]. Tightness was later developed into a theory of cohesion measurement based on slicing [2, 25].

An alternative KSA algorithm is presented in Figure 5. In this approach, the returned value associated with a statement is a real number, rather than simply a value in $\{0, 1\}$. The value assigned to a statement represents the directness of dependence between it and the principal variables of the concept. The weight for statement s is computed as the length of the shortest path from s to a final use vertex of a principal variable, normalized with respect to the length of the longest acyclic path in the program's SDG[2], such that statements with KSA values closer to 1 are more '*key*' and those with KSA values closer to 0 are 'less *key*'. This gives a real value weighting between 0 and 1 for each statement in the concept. The algorithm builds up the function F to be returned, adding a maplet for each statement s_i which maps s_i to its weight.

Observe that, because $Dist(s, s', p)$ is undefined if there is no path from s to s' in the SDG of p, the weight of a statement is also undefined when there is no path from it to the final use vertex of any principal variable. For any such an 'unconnected' statement, the undefinedness of the weight will alert the engineering to a possible anomaly; why is such a statement in a concept if it has no effect on any principal variable? We call this algorithm the 'BallEick' algorithm because it is inspired by Ball and Eick's work on the SeeSlice project[1].

6 CDA Algorithm

An algorithm for producing a weighted Concept Dependency Graph is presented in Figure 6. Weightings will be allocated according to the amount of compu-

[2] In the algorithm, the SDG is used, but there may be analyses for data-intensive programs, for which it would be edifying to consider the replacement of the SDG with the Data Dependence Graph (DDG) and (for control sensitive concepts) to use the Control Dependence Graph (CGD).

```
function KSA_BE(PROGRAM p, CONCEPTSLICE c)
returns: function from STATEMENT to ℝ

let F = {}
let N be the longest acyclic path in the SDG of p
for each s_i in Statements(c)
   for each v_j in PV(c)
      let d_ij = Dist(p, s_i, FinalUse(Statements(c), v_j))
   endfor
   let D_i = min_j d_ij
   let F = F ∪ {s_i ↦ (N − D_i)/N}
endfor
return F
```

Figure 5: Key Statement Analysis 'BallEick' Style

```
function CDA(PROGRAM p, DOMAINMODEL D)
returns: CONCEPTGRAPH

let G = {}
for each c_i ∈ Concepts(p, D)
   for each c_j ∈ Concepts(p, D)   (j ≠ i)
      for each variable v_k in PV(c_j)
         let s_k = Slice(p, {FinalUse(c_j, v_k)})
      endfor
      let Comp = ⋃_k s_k
      let Cont = Comp ∩ Statements(c_i)
      let M = |Cont| / |c_i|
      let G = G ∪ {(c_i, c_j, M)}
   endfor
endfor
return G
```

Figure 6: The Concept Dependency Analysis Algorithm

tation (normalized by concept size) which one concept contributes to the computation of another. To compute this we use an approach based on the slice-based coupling metric of Harman et al. [14]. This approach is a coupling metric, similar to the cohesion metrics of Bieman and Ott [25].

The metric is computed using the principal variables of a concept. The union of slices (restricted to concept c') is then formed. This is the part of c' which contributes to the computation of the principal variables of c. The weight of the edge from c' to c is considered to be the relative amount of c' (normalized by the size of c') which lies in the union of slices. This normalized 'amount of computation' forms a crude way of determining the amount of c' which contributes to the computation denoted by c.

The algorithm starts with an empty graph (G) and goes through each concept (the i loop) adding in weightings from each of the other concepts (the j loop) in the graph. For each pair of concepts, the union of slices on principal variables, *Comp*, is computed and this is used to determine the contribution, *Cont*, that one concept makes to the other. This contribution is reformulated into a metric value between 0 and 1, by calculating its size relative to the size of the whole contributing concept.

7 A Case Study

This section presents a case study which illustrates the application of the four algorithms introduced in the paper. The program concerned (see Figure 8) is based on one drawn from a large financial services organisation and, among other things, calculates mortgage repayments. In the example, we have used a library of 25 concepts and their associated evidence to generate concept bindings and segments.

Suppose that the mortgage products of the organisation are to be overhauled. The legacy system which computes mortgage payments is to be reverse and re-engineered. Specifically, consider the scenario in which an engineer is looking to locate the code which calculates mortgage payments to re-use it (possibly in an amended form) in the re-engineered system.

Thus, the reverse engineer is seeking, initially, to retain the code for calculating mortgage interest, while discarding the remainder of the program. A natural step would be to identify the code which implements mortgage calculations. Unfortunately pure slicing cannot help unless the engineer knows which *variables* are important for this computation. The engineer may be only partially familiar with the code and, therefore, unable to select a suitable variable or set of variables. Concept assignment can be used to produce a set of contiguous statements for which there is evidence that the code performs actions relating to mortgage interest, but the engineer cannot simply extract and reuse this code, since the code sequence is not an executable sub-program. However, by forming the ECS for the `Calculate:MortgageInterest` concept the reverse engineer can extract the code of interest as a executable sub-program.

Selecting the 'calculate mortgage interest' concept produces the concept highlighted by light shading in the left-hand column of Figure 8. Figure 1 depicts the fragment of the domain model used to locate this concept. Using the algorithm in Figure 3 the ECS for calculate mortgage interest additionally identifies the boxed lines shown in the figure. Notice that the line of code

```
MOVE '010' TO APS-RECORD-IN.
```

is not in the ECS, even though it assigns a value to one of the variables (APS-RECORD-IN) referenced by the concept. This is because the value assigned is immediately overwritten by the PERFORM of the code for C00-READ-APS.

The reverse engineer might also analyse the concept using Key Statement Analysis. The principal variables of the concept are:

W-RED-INT-4
W-RED-INT

Using the BiemanOtt style KSA algorithm (Figure 4), the intersection of slices for these two variables consists of the code which computes W-RED-INT-4, since this code is a subset of the code which computes W-RED-INT. We might think of this analysis as revealing a sub-concept (the unrounded result) within the calculate mortgage concept.

Now, suppose instead of applying KSA to the concept, the reverse engineer, instead, chooses to apply it to the ECS. The principal variables of the ECS are:

W-RED-INT-4
OUT-OUTSTANDING
W-RED-INT
APS-RECORD-IN

For these four variables, the intersection of the corresponding slices (Tight) is empty, indicating that no statements are *key* in the ECS according to the BiemanOtt style algorithm. This information is useful, because it indicates that there is no code in the ECS which is germane to all of the computation. This suggests that there may be more than one concept implemented within the boundaries identified by ECS. In this case, the ECS happens to contain a large part of the Read:APSRecord concept and this should probably be separated out. (The possibility of multiple concept binding is discussed briefly as an issue for future work.)

At this point, the reverse engineer might choose to try KSA with a slightly different set of principal variables, based upon the observation that APS-RECORD-IN is an obvious 'odd one out'; it is clearly an input variable (even though it is both global and assigned and, therefore, a 'principal variable' according to Definition 5). For the remaining three variables, the KSA highlights precisely the computation on OUT-OUTSTANDING. That is, the key statements identified are the three boxed statements of the section S10-HOLIDAY-CHECK. This signifies that the flag APS-HOL-MONTH is crucial. Having observed this, the reverse engineer might check to see what the flag APS-HOL-MONTH denotes. A little (human) analysis will reveal that this feature of the system implements 'payment holidays'. This is a product feature aimed at increasing take up and making the product more attractive. It allows the client to skip a payment for one month, by extending the period of payments by one month.

Of course, in the post-overhaul set of products, the payment holiday feature may not be included (or it may be included but behave differently). The identification of the mortgage holiday computation as a set of key statements of the concept alerts the reverse engineer to the importance of this code in determining the mortgage payments and identifies the section of code which needs to be considered.

In Section 5 an alternative, and more fine-grained, KSA algorithm was introduced. This approach uses distance (in the SDG) from the final use vertices of principal variables to determine a weight for a statement, giving a relative measure of *keyness*. To see how this works, consider the concept for mortgage payment. The code segment for the concept is cut out and depicted in Figure 7, along with its CFG and the corresponding SDG for the two principal variables.

Dependence is traced backward from the final use vertices for the two principal variables W-RED-INT-4 and W-RED-INT. The longest acyclic path in the SDG is 6 nodes long (from the final use of W-RED-INT, to 7, 6, 2, 5, 4). Nodes 7 and 6 are only a single edge away from a final use and so the shortest path is length 2. Therefore both nodes receive a KSA value of $\frac{4}{6}$. The shortest path from nodes 5, 2 and 1 is 3 nodes long and so they receive a KSA value of $\frac{3}{6}$. Node 4 is next, with a shortest path of length 4 and a KSA value of $\frac{2}{6}$ and finally node 3 has a KSA value of $\frac{1}{6}$.

The values in themselves are largely immaterial; we can, at best, be measuring on an ordinal scale of 'directness of dependence' [29]. What is important is the order they introduce on nodes. The most key statements are those which define the values of interest (nodes 6 and 7). The next most key are those which directly control the nodes which define the values of interest and those which feed data directly to them. As we move further away from the final use vertices, we reach statements which have a progressively less direct impact upon the computation of the final value of the principal variables. It is this observation which motivates the determination of 'relative keyness' using the 'BallEick' style approach.

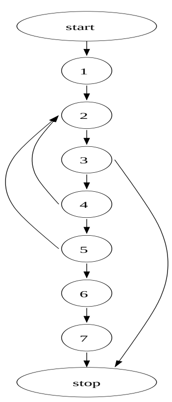
Finally, suppose that the reverse engineer has extracted several concepts[3]. One of the other concepts which is identified is the Write:APSRecord concept shown in the darker shading in the top right-hand column of Figure 8.

The reverse engineer may be interested in the relationship between this concept and the calculate

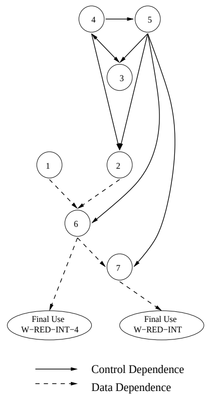
[3] Applying HB-CA to this example actually reveals 10 concepts, but there is insufficient space here to discuss them all in detail.

```
1     PERFORM S10-HOLIDAY-CHECK.
 A00-010.
*    READ APS RECORD
2     PERFORM C00-READ-APS.
3     IF APS-EOF = END-OF-FILE
         GO TO A00-090.
*    CHECK FOR HORIS
4     IF APS-HORIS NOT = 'AH'
         GO TO A00-080.
*    CHECK FOR MORTGAGE INTEREST
5     IF APS-M-INT = ZEROES
         GO TO A00-080.
 A00-020.
*    CALCULATE NEW REDUCED MORTGAGE INTEREST
6     COMPUTE W-RED-INT-4 =
OUT-OUTSTANDING - (W-TAX-RATE * OUT-OUTSTANDING).
7     COMPUTE W-RED-INT ROUNDED = W-RED-INT-4 + 0.
 STOP RUN.
```

Concept for Calculate Mortgage Payment



Control Flow Graph (CFG)



System Dependence Graph (SDG)

Figure 7: Executable Concept Slice for `Calculate:MortgageInterest`

mortgage interest concept. Such a relationship is useful in refining the domain model, which contains inter-concept relationships. It also provides a crude form of assessment of the impact of changes to one concept upon another and the level of 'feature interaction' between concepts. The principal variables of the 'write APS record' concept are:

```
APS-RECORD-OUT
CHECKING-SLIP
```

Using the CDA algorithm of Figure 6, the slice on these two variables contains only one line of the calculate mortgage interest concept:

```
PERFORM C00-READ-APS
```

In computing the relative weight of the edge from the calculate mortgage interest concept to the 'write APS record' concept, we face the familiar issue of how to 'count' lines of code [8, 29]. We have chosen to adopt the (relatively) uncontroversial step of counting Non Comment Source Lines (NCSL). However, as with the determination of principal variables, this choice is a *parameter* to our approach and is adopted here merely for illustration. There are nine NCSLs in the calculate mortgage interest concept and so the weight of the edge from the calculate mortgage interest concept to the 'write APS record' concept is $\frac{1}{9}$.

We have already computed the slice for the principal variables of the calculate mortgage interest concept. The slice was used to form the ECS earlier and consists of the additional boxed lines in Figure 8. We can see that three of these boxed lines are in the 'write APS record' concept. However, only one of them is a NCSL, while there are eight NCSLs in total in the 'write APS record' concept. This gives the weighting of the relationship from the 'write APS record' concept to the calculate mortgage interest concept as $\frac{1}{8}$.

In themselves these figures are relatively meaningless. However, by computing similar weights for all the concepts in the system we obtain a weighted concept dependence graph which can be used to refine our understanding of the domain model and could, for example, form the input to a clustering tool such as Bunch [19, 21, 23].

8 Conclusion

This paper has shown how concept assignment and slicing can be combined to perform unified source code extraction, which extracts code identified by a concept assignment criterion.

The approach has the advantage (over pure concept assignment) that the code extracted is executable, because of the use of slicing to augment the results of concept assignment. It also has the advantage over slicing that the criterion for extraction is expressed at a high level in terms of domain specific and 'meaningful' concepts such as 'master file' and 'update record'. By contrast, pure slicing can only extract subprograms based upon low level criteria — sets of variables.

The paper introduced an algorithm for Executable Concept Slicing (ECS), two algorithms for Key Statement Analysis (KSA) and an algorithm for Concept Dependence Analysis (CDA). The application of these algorithms to reverse engineering was demonstrated using a case study based on a Cobol mortgage calculation program taken from a large financial services company.

9 Future Work

This paper has introduced novel definitions, notation and ideas for unifying concept assignment and slicing. It has presented algorithms and illustrated their application, but there remains much work to be done in order to unlock the full potential of the unified approach.

9.1 Multiple Concept Binding

Currently, HB-CA assumes that there is only a single concept represented in any section of code. This is unrealistic, but it makes the problem more tractable. KSA provides a possible vehicle to analyse the structure of a concept to identify potential split points so that the possible layering of multiple concepts within a rejoin of code can be considered.

9.2 Extending Concept Boundaries

The plausible reasoning approach to concept assignment has been shown to produce good results [9], but, by its very nature, cannot be *guaranteed* to identify the correct concept extent in the code. We have used backward slicing to find the code necessary to make the concept executable. However, if HB-CA has missed some subsequent statements, which only get executed *after* the identified code has been executed, then these statements will form neither a part of the concept nor of the executable concept slice. This does not mean that the ECS will be wrong; it simply means that it will capture only a sub-component of the computation of the concept. To capture the full computation, it may be possible to augment the results of backward slicing with some form of forward slicing [16].

9.3 Concept Clustering Analysis

Concept Dependency Analysis produces a weighted graph of concepts, with the aim of identifying inter-concept relationships. However, a question remains: how can we use the concept dependence graph to determine which concepts are related? This problem is very similar to the software modularization or clustering problem [13, 19, 20]. A tool like BUNCH [19] uses hill-climbing to search for good clusterings of software modules based upon a 'fitness' function. The fitness function is essentially a metric which measures cohesion and coupling between modules in a cluster. The metric has been successfully applied to a number of real-world applications [22, 24]. In all existing applications the module dependence graph used was not weighted, but the metric used does cater for weighted graphs and so there is no reason not to use BUNCH to produce a clustering of concept slices.

10 Acknowledgements

Mark Harman and Rob Hierons are supported, in part, by EPSRC Grants GR/R98938, GR/M58719, GR/M78083 and GR/R43150 and by two development grants from DaimlerChrysler. Nicolas Gold is supported by EPSRC Grant GR/R71733 and would also like to gratefully acknowledge the support of the Computer Sciences Corporation.

References

[1] T. Ball and S. G. Eick. Visualizing program slices. In A. L. Ambler and T. D. Kimura, editors, *Proceedings of the Symposium on Visual Languages*, pages 288–295, Los Alamitos, CA, USA, Oct. 1994. IEEE Computer Society Press.

[2] J. M. Bieman and L. M. Ott. Measuring functional cohesion. *IEEE Transactions on Software Engineering*, 20(8):644–657, Aug. 1994.

[3] T. J. Biggerstaff, B. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *15th International Conference on Software Engineering*, Baltimore, Maryland, May 1993. IEEE Computer Society Press, Los Alamitos, California, USA.

[4] D. W. Binkley. Computing amorphous program slices using dependence graphs and a data-flow model. In *ACM Symposium on Applied Computing*, pages 519–525, The Menger, San Antonio, Texas, U.S.A., 1999. ACM Press, New York, NY, USA.

[5] G. Canfora, A. Cimitile, and M. Munro. RE^2: Reverse engineering and reuse re-engineering. *Journal of Software Maintenance : Research and Practice*, 6(2):53–72, 1994.

[6] A. Cimitile, A. R. Fasolino, and P. Marascea. Reuse reengineering and validation via concept assignment. In *Proceedings of the International Conference on Software Maintenance 1993*, pages 216–225. IEEE Computer Society Press, Sept. 1993.

[7] P. Devanbu, R. J. Brachman, P. G. Selfridge, and B. W. Ballard. LaSSIE: A knowledge-based software information system. *Communications of the ACM*, 34(5):35–49, May 1991.

[8] N. E. Fenton. *Software Metrics: A Rigorous Approach*. Chapman and Hall, 1990.

[9] N. E. Gold. *Hypothesis-Based Concept Assignment to Support Software Maintenance*. PhD Thesis, Department of Computer Science, University of Durham, 2000.

[10] N. E. Gold. Hypothesis-based concept assignment to support software maintenance. In *IEEE International Conference on Software Maintenance (ICSM'01)*, pages 545–548, Florence, Italy, Nov. 2001. IEEE Computer Society Press, Los Alamitos, California, USA.

[11] N. E. Gold and K. H. Bennett. A flexible method for segmentation in concept assignment. In 9^{th} *IEEE*

International Workshop on Program Comprehension (IWPC'01), pages 135–144, Toronto, Canada, May 2001. IEEE Computer Society Press, Los Alamitos, California, USA.

[12] M. Harman and S. Danicic. Amorphous program slicing. In 5^{th} *IEEE International Workshop on Program Comprenhesion (IWPC'97)*, pages 70–79, Dearborn, Michigan, USA, May 1997. IEEE Computer Society Press, Los Alamitos, California, USA.

[13] M. Harman, R. Hierons, and M. Proctor. A new representation and crossover operator for search-based optimization of software modularization. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1351–1358, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[14] M. Harman, M. Okunlawon, B. Sivagurunathan, and S. Danicic. Slice-based measurement of coupling. In R. Harrison, editor, 19^{th} *ICSE, Workshop on Process Modelling and Empirical Studies of Software Evolution*, Boston, Massachusetts, USA, May 1997.

[15] R. M. Hierons, M. Harman, C. Fox, L. Ouarbya, and M. Daoudi. Conditioned slicing supports partition testing. *Software Testing, Verification and Reliability*, 12:23–28, Mar. 2002.

[16] S. Horwitz, T. Reps, and D. W. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[17] V. Karakostas. Intelligent search and acquisition of business knowledge from programs. *Journal of Software Maintenance: Research and Practice*, 4:1–17, 1992.

[18] H. D. Longworth, L. M. Ott, and M. R. Smith. The relationship between program complexity and slice complexity during debugging tasks. In *Proceedings of the Computer Software and Applications Conference (COMPSAC'86)*, pages 383–389, 1986.

[19] S. Mancoridis, B. S. Mitchell, Y.-F. Chen, and E. R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings; IEEE International Conference on Software Maintenance*, pages 50–59. IEEE Computer Society Press, 1999.

[20] S. Mancoridis, B. S. Mitchell, C. Rorres, Y.-F. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *International Workshop on Program Comprehension (IWPC'98)*, pages 45–53, Ischia, Italy, 1998. IEEE Computer Society Press, Los Alamitos, California, USA.

[21] S. Mancoridis, T. S. Souder, B. S. Mitchell, Y.-F. Chen, and E. R. Gansner. REPortal: A web-based portal site for reverse engineering. In 8^{th} *Working Conference on Reverse Engineering*, pages 221–230, Stuttgart, Oct. 2001. IEEE Computer Society Press, Los Alamitos, California, USA.

[22] B. S. Mitchell. *A Heuristic Search Approach to Solving the Software Clustering Problem.* PhD Thesis, Drexel University, Philadelphia, PA, Jan. 2002.

[23] B. S. Mitchell and S. Mancoridis. REPortal: A web-based portal site for reverse engineering. In 8^{th} *Working Conference on Reverse Engineering*, pages 93–102, Stuttgart, Oct. 2001. IEEE Computer Society Press, Los Alamitos, California, USA.

[24] B. S. Mitchell and S. Mancoridis. Using heuristic search techniques to extract design abstractions from source code. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1375–1382, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[25] L. M. Ott and J. M. Bieman. Program slices as an abstraction for cohesion measurement. In M. Harman and K. Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 681–699. Elsevier, 1998.

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

[27] L. M. Ott and J. J. Thuss. Slice based metrics for estimating cohesion. In *Proceedings of the IEEE-CS International Metrics Symposium*, pages 71–81, Baltimore, Maryland, USA, May 1993. IEEE Computer Society Press, Los Alamitos, California, USA.

[28] C. Rich and R. C. Waters. *The Programmer's Apprentice.* ACM Press (Frontier Series), 1990.

[29] M. J. Shepperd. *Foundations of software measurement.* Prentice Hall, 1995.

[30] M. Ward. The formal approach to source code analysis and manipulation. In 1^{st} *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 185–193, Florence, Italy, 2001. IEEE Computer Society Press, Los Alamitos, California, USA.

[31] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[32] L. M. Wills. *Automated Program Recognition by Graph Parsing.* PhD Thesis, AI Lab, Massachusetts Institute of Technology, 1992.

```
:
PROCEDURE DIVISION.
A00-CONTROL SECTION.
*   INITIAL PROCESSING
A00-000.
    PERFORM S10-HOLIDAY-CHECK.
    MOVE '01' TO DL-INPUT-FORMAT.
    CALL 'DATEPRES' USING DATE-LINKAGE-PARMS.
    MOVE DL-OUT-DD-MM-CCYY TO H1-DATE.
    MOVE SPACES TO CHECKING-SLIP.
    MOVE '011' TO APS-RECORD-OUT.
    CALL 'GBAAZ0X' USING APS-RECORD-OUT.
    CALL 'GBABB0X' USING CHECKING-SLIP.
    MOVE '010' TO APS-RECORD-IN.

A00-010.
*    READ APS RECORD
     PERFORM C00-READ-APS.
     IF APS-EOF = END-OF-FILE
           GO TO A00-090.
*    CHECK FOR HORIS
     IF APS-HORIS NOT = 'AH'
           GO TO A00-080.
*    CHECK FOR MORTGAGE INTEREST
     IF APS-M-INT = ZEROES
           GO TO A00-080.

 A00-020.
*    CALCULATE NEW REDUCED MORTGAGE INTEREST
     COMPUTE W-RED-INT-4 =
OUT-OUTSTANDING - (W-TAX-RATE * OUT-OUTSTANDING).
     COMPUTE W-RED-INT ROUNDED = W-RED-INT-4 + 0.
    IF GBAIA110 = 'M'
          MOVE 12 TO W-FREQ
          MOVE 0.12 TO W-FREQ-P.
    IF GBAIA110 = 'Q'
          MOVE 4 TO W-FREQ
          MOVE 0.03 TO W-FREQ-P.
    COMPUTE W-RED-INT-2 = W-RED-INT / W-FREQ.
    SUBTRACT 0.0005 FROM W-RED-INT-2.
    COMPUTE W-RED-INT-3 ROUNDED = W-RED-INT-2 + 0.

A00-030.
    MULTIPLY W-FREQ BY W-RED-INT-3.
    MOVE W-RED-INT-3 TO GBAOA191.
    EJECT
    IF GBAIA190 = SPACES
          GO TO A00-040.
    IF GBAIA191 = ZEROES
          GO TO A00-040.
    DIVIDE GBAOA191 BY GBAIA191 GIVING W-PERCENTAGE.
    IF W-PERCENTAGE GREATER THAN 1.03
          GO TO A00-040.
    IF W-PERCENTAGE LESS THAN 0.97
          GO TO A00-040.
    GO TO A00-070.

A00-040.
    PERFORM C20-PRINT.

A00-070.
    MOVE SPACES TO CHECKING-SLIP.
    MOVE GBAIA010 TO CS-POLICY.
    MOVE '2' TO CS-TYPE.
    MOVE GBAIA019 TO CS-STANDARD (1).
    MOVE GBAOA019 TO CS-STANDARD (2).
    CALL 'GBABB0X' USING CHECKING-SLIP.
```

```
A00-080.
    PERFORM C10-WRITE-APS.
    GO TO A00-010.

 A00-090.
    MOVE '3' TO W-GBCM0133-2.
* END OF JOB PROCESSING
    CALL 'GBCM0133' USING APS-RECORD-IN
          W-GBCM0133-2
          W-GBCM0133-3.
    MOVE END-OF-FILE TO APS-RECORD-OUT.
    CALL 'GBAAZ0X' USING APS-RECORD-OUT.
    MOVE END-OF-FILE TO CHECKING-SLIP.
    CALL 'GBABB0X' USING CHECKING-SLIP.
A00-999.
    STOP RUN.
    EJECT
C00-READ-APS SECTION.
C00-000.
*   READ APS MASTER FILE
    CALL 'GBAAY0X' USING APS-RECORD-IN.
    IF APS-EOF = END-OF-FILE
          MOVE HIGH-VALUES TO APS-RECORD-IN.
C00-999.
    EXIT.
    SKIP3
C10-WRITE-APS SECTION.
*   WRITE APS MASTER FILE
    MOVE '2' TO W-GBCM0133-2.
    CALL 'GBCM0133'
    USING APS-RECORD-OUT W-GBCM0133-2.
    CALL 'GBAAZ0X' USING APS-RECORD-OUT.
C10-999.
    EXIT.
    SKIP3
C20-PRINT SECTION.
C20-000.
    IF A-LINENO LESS THAN 25
          GO TO C20-010.
    ADD 1 TO A-PAGENO.
    MOVE A-PAGENO TO H1-PAGE.
    MOVE C-1 TO P-CC.
    MOVE H1-HEADLINE TO P-LL.
    PERFORM S00-PRINT.
    MOVE WS-2 TO P-CC.
    MOVE H1-HEADLINE TO P-LL.
    PERFORM S00-PRINT.
    MOVE 0 TO A-LINENO.

C20-010.
    MOVE WS-2 TO P-CC.
    MOVE GBAIA010 TO P1-KEY.
    MOVE P1-DATALINE TO P-LL.
    PERFORM S00-PRINT.
    MOVE SPACES TO P-LL.
    ADD 2 TO A-LINENO.
C20-999.
    EXIT.
    EJECT
S00-PRINT SECTION.
S00-000.
*   PRINTS A LINE
    CALL 'PRINT' USING P-PRINTLINE.
S00-999.
    EXIT.
S10-HOLIDAY-CHECK SECTION.
*   CHECK FOR PAYMENT HOLIDAY
    IF APS-HOL-MONTH = DL-MONTH
          MOVE 'Y' TO OUT-PAYMENT-HOL
          MOVE ZEROES TO OUT-OUTSTANDING.
S10-999.
    EXIT.
```

Key:

Dark Shaded: `Write:APSRecord` concept

Light Shaded: `Calculate:MortgageInterest` concept

Boxed: Extra code in ECS for `Calculate:MortgageInterest`

Figure 8: Cobol Mortgage Payment Calculation Program