Unifying program slicing and concept assignment for higher-level executable source code extraction

N. E. Gold*[1], M. Harman[2], D. Binkley[3], R. M. Hierons[2]

[1] *Information Systems Group, Department of Computation, UMIST, PO Box 88, Sackville Street, Manchester, M60 1QD, U.K.*
[2] *Brunel University, Uxbridge, Middlesex, UB8 3PH, U.K.*
[3] *Loyola College, Baltimore, MD 21210-2699, U.S.A.*

SUMMARY

Both program slicing and concept assignment have been proposed as source code extraction techniques. Unfortunately, each has a weakness which prevents wider application. For slicing, the extraction criterion is expressed at a very low level; constructing a slicing criterion requires detailed code knowledge which is often unavailable. The concept assignment extraction criterion is expressed at the domain level. Howeverm, unlike a slice, the extracted code is not executable as a separate subprogram in its own right.

This paper introduces a unification of slicing and concept assignment which exploits their combined advantages, while overcoming these two individual weaknesses. Our 'concept slices' are executable programs extracted using high level criteria. The paper introduces four techniques that combine slicing and concept assignment and algorithms for each.

These algorithms were implemented in two separate tools used to illustrate the application of the concept slicing algorithms in two very different case studies. The first is a commercially–written COBOL module from a large financial organization, the second is an open source utility program written in C.

KEY WORDS: *Program Slicing; Concept Assignment; Reverse Engineering*

*Correspondence to: Nicolas Gold, Information Systems Group, Department of Computation, UMIST, PO Box 88, Sackville Street, Manchester, M60 1QD, U.K.

Introduction

In order to mitigate the high costs involved in maintaining software systems, it is beneficial to extract executable sub-components for use in many software engineering activities such as program comprehension and reverse engineering. Ideally, these subcomponents would be extracted automatically based on high-level criteria. Furthermore, the extracted code would be semantically related to the original so it could be used in the latter's stead. An example might be to extract a sub-component undertaking, for example, the same file update operation or reactor shutdown procedure. This problem can be formalized as follows:

> Given a program, construct the simplest program that performs a particular aspect of the original program's functionality.

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 as output. Program slicing [42] uses control and data dependencies in the code to determine the statements that affect a chosen computation. The utility of program slicing comes from its ability to automatically extract an executable portion of a program guaranteed to be semantically equivalent to the original with respect to the *slicing criterion*.

Originally, as formulated by Weiser [42], slicing was static and syntax–preserving. More recently dynamic [31], amorphous [22, 25] and conditioned [10, 12] forms of slicing have been introduced. there are several surveys of slicing [7, 9, 13, 24, 41] which describe its applications, implementation and empirical results. This paper is concerned solely with static slicing.

Concept assignment techniques aim to relate information about the problem and software engineering domains to portions of source code. Our focus is on the Hypothesis-Based Concept Assignment (HB-CA) method [19], which falls into the category of plausible-reasoning techniques. Such techniques tend to be linear in their computational cost [4] and provide a 'best-guess' at the concept being implemented by a particular section of code. HB-CA uses a simple domain model and primarily informal information (*e.g.*, identifiers, comments) to map concepts to sections of source code. A domain-model concept and the possible evidence associated with it form the criterion used for concept assignment. The result of HB-CA typically takes the form of a contiguous region of source code that is not executable.

Program slicing has the advantage of extracting independently executable (sub)programs, but is hampered by the low level nature in which the computation of interest is specified by the slicing criterion (a variable and a program point). HB-CA has the advantage of using criteria at an appropriate level (*e.g.*, 'master file', 'error recovery', or 'log update') but is hampered by the non-executability of the result. Therefore, if the two techniques could be combined, each would provide a solution to the principal deficiency of the other.

In this paper, we show how program slicing can be combined with HB-CA. Specifically, the following contributions are made:

- the introduction of a framework for combining program slicing with Hypothesis-Based Concept Assignment
- the introduction of four techniques that fit into this framework

  - Executable Concept Slicing
  - Forward Concept Slicing
  - Key Statement Analysis
  - Concept Dependency Analysis

- algorithms for each of the techniques
- case studies that illustrate the implementation of these algorithms as applied to two very different problems: a COBOL module from a commercial financial system and an open-source utility in C.

The remainder of this paper is organized as follows: The next section presents background on program slicing and concept assignment. This is followed by the four technical contributions of this paper: first, the framework for unifying HB-CA and program slicing, second, the techniques for combining them, and third an implementation of each, and, fourth, two case studies that explore the new techniques. Finally, we draw the material together and conclude.

Background

This section provides background on program slicing and concept assignment. Program slicing [43] is defined with respect to a 'slicing criterion', typically a set of program variables and a program location. However, in this paper, a slicing criterion will consist solely of a nodes from a program's control-flow graph. The set of program variables is assumed to be the variables referenced at each node. Two kinds of slices are considered: backward and forward.

Definition 1 (Backwards Slice [6])
A *backward slice* of a program p for the slicing criterion S, denoted by *Slice(p, S)*, is an executable subprogram, s, that consists of those statements and predicates of p that transitively affect the computation represented by the elements of S, such that s behaves identically to p with respect to the sequence of values computed at each of the statements represented by S.

Definition 2 (Forward Slice [27])
A *forward slice* of a program p for the slicing criterion S, denoted by *ForwardSlice(p, S)*, consists of those statements and predicates of p that are transitively affected by the computation represented by the elements of S.

Backward and forward slices can be computed from a program's System Dependence Graph (SDG) as the solution of a graph reachability problem [27]. The vertices of an SDG are essentially those of the program's control-flow graph. Its edges represents control and data-flow dependences between the vertices. The SDG also contains 'final use' vertices, denoted as *FinalUse(p,v)*, for each variable v used in program p. These vertices allow slices to be constructed with respect to the final value of a variable.

In addition to program slicing, concept slicing makes use of concept assignment. The concept assignment[†] problem was defined by Biggerstaff et al. 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 [4].’ Concept assigners can be used for a variety of tasks *e.g.*, program comprehension [4, 19, 28], validating the adequacy of a candidate criterion when identifying reusable modules [11], and identifying criteria for program slicing as described here.

Biggerstaff et al. classify the intelligent agents (tools) that can undertake the concept assignment task thus [4]:

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 [14].
2. Plan driven, algorithmic program understanders or recognisers. Two examples of this type are the Programmer’s Apprentice [39], and GRASPR [44].
3. Model driven, plausible reasoning systems. Examples of this type include DM-TAO [4, 5], IRENE [28], and HB-CA [17, 19].

Furthermore, they claim that systems using the first 2 approaches are good at completely deriving concepts within small-scale programs but cannot deal with large-scale programs due to overwhelming computational growth. In contrast, the third approach can easily handle large-scale programs since its computational growth appears to be linear in the length of the program under analysis [4]. This category is of particular interest because such systems tend to be scalable and are theoretically capable of assigning higher-level concepts than systems from approaches 1 and 2 [4].

Concept assignments are based on evidence available in the code being analysed from which a ‘best guess’ is taken. The results are thus based on plausible reasoning (rather than deductive reasoning as used by category 2 systems) and frequently employ informal or domain-oriented source code information in their analysis.

Informal information in source code has been used in concept recovery by a number of approaches. The DM-TAO system [4, 5] adopted a rich semantic network as its representational and analytical engine. It relates source code evidence at the bottom layer to concepts at the top by propagating weights from the node that had been triggered. This provides strong analytical power but at a potentially high cost since the network may be hard to construct or maintain. The IRENE system [28] aimed to discover business rules in COBOL by matching informal information in the source code to parts of formal structures in the knowledge base representing business rules, thus enabling generic business rules to be related to their instances in a program. This is targeted at recovering a more specific type of concept than the other approaches discussed here. Recent work has focussed on the use of feed-forward and recurrent neural networks for analyzing informal source code information [34]. In this approach, a domain

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

expert constructs a concept taxonomy, to which source code evidence in the form of identifiers and comments is related. This process trains a neural network that can then be used to analyse other source code in the domain. The results showed that using these neural networks produced a much higher concept recognition rate than lexical matching alone. Other approaches have used concept recovery techniques to perform component clustering based on file names [1] and latent semantic indexing [33]. This paper is concerned with the plausible-reasoning category, and it is to this approach that we now turn our attention.

Hypothesis-Based Concept Assignment (HB-CA)

Hypothesis-Based Concept Assignment [17, 18, 20, 19] is one of the most recent examples of a plausible-reasoning concept assignment approach. It addresses the first part of Biggerstaff et al.'s concept assignment problem (recognising concepts and relating them to portions of the program).

As is typical of plausible-reasoning approaches, HB-CA employs a knowledge-base (termed the 'library') to drive its analysis. The knowledge-base contains concepts (terms of interest nominated by the maintainer [19]) which are either actions (concepts describing an activity *e.g.*, Read) or objects (concepts describing something that is acted on *e.g.*, File) and objects can either be primary or secondary. The latter designation simply determines a concept's place in the simple hierarchy (*e.g.*, File might be a primary concept and MasterFile might be secondary; a simple, informal, inheritance mechanism). Concepts can be composed in an Action:Object structure (with any secondary objects composed by implication). Objects can be specialised using the primary and secondary mechanism described above.

Concepts are linked to indicators which are designated evidence that, if found in source code, may indicate the presence of the concept. Indicators are divided into classes (identifier, keyword, and comment) based on the type of token they are intended to match. In practice, although this classification is enforced, it makes little difference to the assignments since, in most cases, all three classes are always used. The library is created by a software maintainer in advance of HB-CA being applied. Figure 1 shows an example fragment from a library.

The HB-CA method is fully automatic (after the creation of the knowledge base) and has three stages.

Stage 1 is Hypothesis Generation. This involves tokenising the source code (including comments) and comparing each token to all of the library indicators. Options for matching indicators include case sensitivity and substring matching. When a match is found, a hypothesis for the appropriate concept is generated at the indicator's position in the source code. The result of stage 1 is therefore an ordered list of hypotheses for concepts.

Stage 2 is Segmentation. In this stage, the hypothesis list is divided into segments, initially using the subroutine boundaries of the program. Such segment boundaries are termed 'hard' as they relate to the syntactic structure of the program. Since HB-CA is designed to work on monolithic code or large subroutines, it may not be helpful to simply assign the most likely concepts on the basis of one-per-subroutine. To increase the resolution of the analysis, each hard segment is examined to determine whether it may be possible to form clusters of conceptual focus within the segment. If the potential exists to form such clusters, a Self-Organising Map (SOM) [29] is used to perform an unsupervised clustering of concepts (note: clustering takes

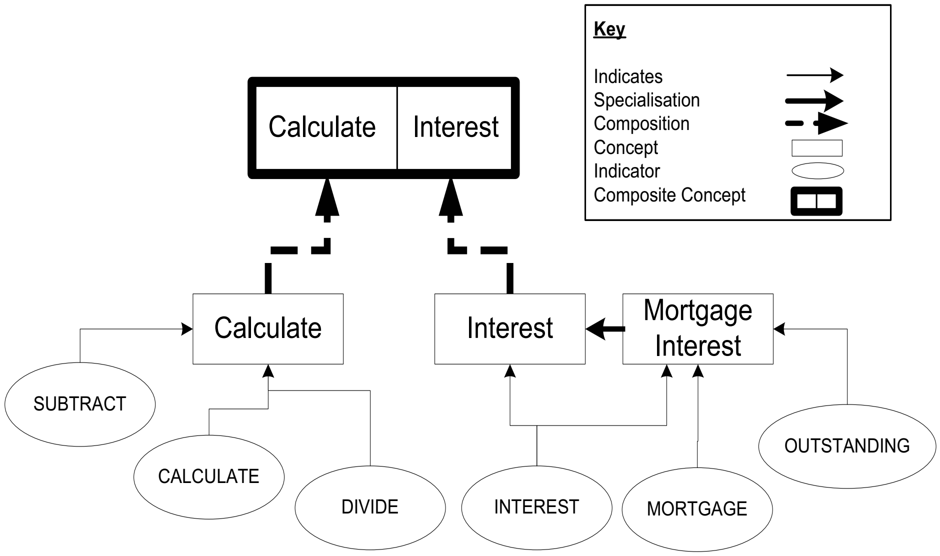


Figure 1. A Fragment of a Domain Model

place in concept-space rather than syntax-space). The results of SOM clustering are analysed and, if appropriate, soft segments are created to replace the original hard segment. In this way concepts can be linked to those sections of source code that implement them, regardless of size, but without crossing subroutine boundaries. A more detailed treatment of the segmentation process can be found in Gold's work [17, 20]. The result of stage 2 is a segmented hypothesis list.

The final stage is Concept Binding. Here, the hypotheses in each segment are analysed to determine which occurs most frequently. The general principle is that the most frequently occurring hypothesis indicates the most likely concept for the segment. A number of disambiguation rules are applied to select a concept where several different hypotheses occur equally frequently. The result of this stage (and thus the result of the overall method) is a series of concept bindings linking regions of source code (bounded by the lines on which the indicators of the outer hypotheses of each segment occur) to a library concept.

HB-CA was originally developed to aid maintainers working on COBOL programs (specifically IBM COBOL II, although this particular dialect has no special features that make HB-CA more applicable to it than any other dialect of COBOL). It was designed to focus on what happens in a program rather than the objects to which it happens. Therefore, action concepts take precedence over objects (indeed, objects are not allowed to be bound to a region of code without an associated action; the reverse is permitted however). As a consequence, the `DATA DIVISION` (*i.e.*, all variable declarations) is not processed, but analysis begins at the `PROCEDURE DIVISION`. HB-CA was evaluated on programs drawn from a financial services system and performed well in the case studies undertaken (see [17, 19]).

The original case studies were undertaken using an implementation in Borland Delphi termed HB-CAS [17] (SOM processing was (and is) performed using Kohonen's SOM_PAK system [30]). In order to aid interoperability, HB-CAS has since been rewritten (also in Delphi) as a system called WeSCA. Data exchange between stages is now performed using GXL and each stage can be invoked separately. The package consists of several programs. The main GUI co-ordinates the analysis, allowing the selection of a library coded in a GXL, the file for analysis, options for the analysis (*e.g.*, minimum acceptable thresholds for recognition), and the target language. Analysis can be undertaken in one step, or by running each stage separately. The GXL output from each stage can be viewed or saved. The entire output set can also be saved post-analysis, or rendered as HTML for viewing the source code with highlighted concepts. A command-line interface version of the system can also be used. This has the same core code but is wrapped in an interface more amenable to script processing. The package also includes a library editor for managing concepts and indicators. This exports a library definition to GXL.

For the case studies reported here, we executed WeSCA to obtain the concept bindings and exported the results to HTML. Those regions of code highlighted in the HTML were then highlighted in CodeSurfer to form the criteria for the concepts. In future we intend to integrate these packages more closely to provide end-to-end analysis.

In order to apply HB-CA to the C case study reported in this paper, a hypothesis generator capable of analysing C source code was needed. Fortunately, the HB-CA method was designed to be as language-independent as possible. After the token extraction stage of hypothesis generation, no further reference is made to the source code until concepts are finally bound at the end of the process. The task of adapting HB-CA to analyse C thus involved creating a tokeniser for C. The tokeniser extracts tokens under the following conditions:

1. header comments are skipped (consistent with COBOL analyser), and
2. all declarations and function prototypes are skipped until the first full function definition. From this point, all code is treated as procedural.

This results in a weak parse (*e.g.* as string literals can be extracted as identifiers) but since we use all classes of indicator it has little effect on the overall result.

Early experiments with the new lexical analyzer showed that the 'standard' substring analysis used by WeSCA caused significant problems later in the process. Normally, WeSCA designates a match if the token string exists in the library indicator string or vice versa. This works well in COBOL where variable names are often several characters long. However, in C, the prevalence of single character variable names caused matches to be found with almost every library indicator causing confusion for the concept binder. Consequently, uni-directional substring matching was implemented and has been used throughout this case study. Matches are designated only if the strings match exactly or if the library indicator string occurs in the source token.

The modified WeSCA system was evaluated on the SOM_PAK source code with a generic library of concepts for the C language. It was scored in a similar way to the COBOL II version (post-process analysis of the validity of system's output by the authors) and performed at an

Table I. Overview of the Concept Slicing Framework

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

acceptable level. That is, 27 of the 37 concepts found were judged accurate (the nominated code matches the specified concept at some point) with 14 of these judged strictly accurate (the nominated code is primarily concerned with the specified concept). These results are similar to those found in previous evaluations (see [17, 19] for more detail on the evaluation methods and criteria).

We now introduce the notation used later to denote concept bindings made by HB-CA.

Definition 3 (Concept Binding)
A concept binding 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 binding c, *Tag(c)* refers to the name of the concept binding c, while *Statements(c)* refers to its statements.

Definition 4 (Concepts)
For a program p and domain model D, *Concepts(p,D)* refers to the set of all concepts assigned to p according to D.

Unifying Framework

This section presents a framework for concept slicing, which combines slicing and concept assignment. Four new techniques are introduced: Executable Concept Slicing, Forward Concept Slicing, Key Statement Analysis, and Concept Dependency Analysis. Table I presents an overview of the four concept slicing techniques, and includes potential applications for each. In the table, inter-concept algorithms are those that undertake analysis outside and between concepts, while intra-concept algorithms are those where the analysis is focussed within a concept.

Executable Concept Slicing (ECS)

Executable Concept Slicing is the basis of the approach we advocate. An ECS is formed using slicing to augment the results of HB-CA to make a concept binding (or bindings) into an executable (sub)program. Taking the variables occurring in the nominated concept binding as a slicing criterion, a backward slice is formed that incorporates all statements relevant to the computation of these variables.

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

Forward Concept Slicing (FCS)

ECS uses backward slicing to capture computations related to the concept bindings. For certain applications, (*e.g.*, the detection of ripple effects) forward slicing is more appropriate. Forward slicing finds code affected by a computation, rather than the code that affects a computation. Forward Concept Slicing (FCS) operates in the same manner to ECS except it uses a forward slice [27] to capture code that could be affected by a concept.

More formally, an algorithm for FCS is a function that takes a program, p, and a domain model, D, and produces a set of sub-components, one per concept binding in the program p according to D. As a forward slice is not guaranteed to be executable, the resulting forward concept slice may not be executable. In forming an FCS, the set of statements tagged with the concept name may no longer be contiguous.

Key Statement Analysis (KSA)

Key Statement Analysis (KSA) is a refinement of ECS (but presently not FCS). It is based on the observation that some statements in a concept binding or concept slice contribute more to the computation embodied by the binding than others. We term these statements 'key statements'. If such statements can be identified, we can use them to increase the richness of information provided to a software engineer. It may help in the extraction of a tighter subset of code for reuse, to focus attention more rapidly on the part of a computation most likely to cause a problem, or to identify poor cohesion in the source code being studied.

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

At least two approaches to KSA are envisaged. The first, simpler, approach would merely identify a subset of the statements as being key (*i.e.*, a binary decision). The second, more elaborate, approach would assign weights to each statement to indicate relative key*ness*. These weightings will be real numbers in the range 0 to 1. The simple approach thus can be seen as a special case of the elaborate approach in which the only two outcomes are 0 and 1.

Concept Dependency Analysis (CDA)

The final technique we define, Concept Dependency Analysis (CDA), is aimed at improving the analysis itself. To use HB-CA, an initial domain model must be created by a software engineer using their knowledge and experience of both the application and software engineering domains. This model can be reused across applications to train new engineers. Apart from guidance based on anecdotal evidence [17], there is currently no way to aid the software engineer in improving the domain model. He or she must manually undertake observations of factors such as concepts that occur together frequently, or concepts that could be linked by the data or control flow dependence through a concept slice. The domain model is thus only improved in a poorly-defined, ad-hoc manner. A clearly defined and tool-assisted feedback mechanism would greatly improve the disciplined and systematic evolution of the domain model. This should facilitate process improvement and better performance with successive applications of HB-CA.

More formally, an algorithm for CDA is a function that takes a program, p, and a domain model D, and returns a *Concept Dependence Graph* (CDG), which indicates the strength of association between concepts discovered in the program. A CDG is a directed graph in which the nodes are concept bindings and the edges are weighted. Thus, the CDG is a set of triples, such that triple *(c, c', w)* is in the graph iff there is an edge from concept binding *c* to *c'* with weight w.

Algorithms

This section presents algorithms for each of the four techniques described above. The algorithms for KSA and CDA are based on the notion of *principal variables*. The principal variables are those which might be considered to be the result of a set of statements [3, 35]. Since the set of statements in this case would be a concept binding, the principal variables might be considered to be the result of that particular concept. Bieman and Ott indicate, in their work on slice-based cohesion measurement [32, 36], that deciding which variables should be nominated as principal is somewhat arbitrary; changing this definition will alter the results of the algorithms by which it is used. The definition of a principal variable is thus effectively a parameter of the concept slicing approaches described in this paper. The definition below is derived from Bieman and Ott [3] and is used as a working definition for the case studies in this paper.

Definition 5 (Principal Variable)
A variable v in a set of statements S is a principal variable, denoted by $v \in PV(S)$, iff it is

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

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

**let** $\{c_1, \ldots, c_n\} = Concepts(p, D)$
**for** each $c_i \in \{c_1, \ldots, c_n\}$
   **let** $ECS_i = Slice(p, Statements(c_i))$
   **let** $Tag(ECS_i) = Tag(c_i)$
**endfor**
**return** $\{ECS_1, \ldots, ECS_n\}$
```

Figure 2. The Executable Concept Slicing Algorithm

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

**let** $\{c_1, \ldots, c_n\} = Concepts(p, D)$
**for** each $c_i \in \{c_1, \ldots, c_n\}$
   **let** $FCS_i = ForwardSlice(p, Statements(c_i))$
   **let** $Tag(FCS_i) = Tag(c_i)$
**endfor**
**return** $\{FCS_1, \ldots, FCS_n\}$
```

Figure 3. The Forward Concept Slicing Algorithm

To begin with, the algorithms for ECS and FCS are shown in Figures 2 and 3, respectively. As stated above, the statements and variables of the nominated concept binding form the criterion on which slices are computed. For ECS, statements in the union of the backward slices form the concept slice, which has the same behavior as the original program when attention is focused on the statements of the concept. The FCS algorithm is identical to the ECS algorithm except for the use of forward slicing rather than backward slicing.

Figures 4 and 5 present two KSA implementations. Both make use of the set of principal variables of a concept binding to form a set of slices. The intersection of these slices contains the statements that contribute to the computation of every principal variable. These are the statements *key* to the concept.

The first KSA algorithm KSA$_{BO}$ is inspired by Bieman and Ott's work on measuring cohesion using slicing [3, 32, 36, 37, 38] and the 'Tightness' metric introduced by Ott and Thuss [37]. This metric measures the proportion of a function in the intersection of all slices on principal variables relative to the size of the function itself. For example, a tightness value of 1 indicates that all of the slices taken with respect to the principal variables include all of the function. The function KSA$_{BO}$ takes a program and a concept binding within it, and returns a function that describes the relative weight of each statement in the concept binding. In this case, the result is either 0 or 1, with 1 signifying that a statement is a key statement and 0 signifying that it is not.

The second KSA algorithm (KSA$_{BE}$), inspired by Ball and Eick's work on the SeeSlice project [2], makes use of the notion of dependence distance:

Definition 6 (SDG Distance)
Given statements *s* and *s'* of a program, *p*, the distance between them, denoted by *Dist(p,s,s')* is the length of the *shortest* path from *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.

With KSA$_{BE}$, 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 binding. The weight for a 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[‡]. Statements with KSA values closer to 1 are more '*key*' than those with KSA values closer to 0. The algorithm from Figure 5 builds up the function *F* to be returned, adding a maplet for each statement s_i which maps s_i to its weight.

Finally, Figure 6 presents the CDA algorithm. Weightings between concepts are assigned according to the amount of computation (normalised by concept size) which one concept contributes to the computation of another. We use an approach based on the slice-based coupling metric of Harman et al. [26], and similar to the cohesion metrics of Bieman and Ott [36].

In more detail, the metric is computed between two concept bindings *c* and *c'* using the principal variables of *c*. First, the union of the slices taken with respect to the principal variables of *c* is intersected with the statements of *c'*. This captures the part of *c'* that contributes to the computation of the principal variables of *c*. The relative overlap (normalised by the size of *c'*) is the weight of the edge from *c'* to *c*. This forms a crude metric for of determining the amount of *c'* that contributes to the computation of *c*. In terms of the goal of CDA (*i.e.*, the improvement of the domain model), the more computation that a concept binding contributes to another, the more likely it is that these two concepts are related.

Case Studies

This section describes two case studies undertaken with implementations of our algorithms. The first concerns a commercial COBOL program that computes mortgage repayments. The

[‡] In this algorithm we use the SDG, but it would be appropriate to consider replacing this with the Data Dependence Graph or Control Dependence Graph for data-intensive programs or control-sensitive concepts respectively.

```
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 $= \bigcap_i s_i$
let KS $= Statements(c) \cap Tight$
return $\lambda x.$ if $x \in KS$ then 1 else 0
```

Figure 4. Key Statement Analysis 'Bieman-Ott' Style

```
function $KSA_{BE}$(PROGRAM p, CONCEPTSLICE c)
returns: function from STATEMENT to $\mathbb{R}$

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 $F = F \cup \{s_i \mapsto \frac{N - D_i}{N}\}$
endfor
return $F$
```

Figure 5. Key Statement Analysis 'Ball-Eick' Style

second concerns an open source C program that computes user accounting information. Together these two case studies illustrate the effectiveness of concept slicing.

Case Study 1: Mortgage Repayments

The program in this case study, as shown in Figure 8, is based on one drawn from a large financial services organization 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. This library is slightly extended version of that presented by Gold [17].

Suppose that the mortgage products of an organization are to be overhauled. The legacy system which computes mortgage payments is to be reverse and then re-engineered. Specifically,

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

let $G = \{\}$
for each $c_i \in Concepts(p, D)$
   for each $c_j \in Concepts(p, D) \quad (j \neq i)$
      for each variable $v_k$ in $PV(c_j)$
         let $s_k = Slice(p, \{FinalUse(c_j, v_k)\})$
      endfor
      let $Comp = \bigcup_k s_k$
      let $Cont = Comp \cap Statements(c_i)$
      let $M = \frac{|Cont|}{|c_i|}$
      let $G = G \cup \{(c_i, c_j, M)\}$
   endfor
endfor
return $G$
```

Figure 6. The Concept Dependency Analysis Algorithm

consider the scenario in which an engineer is looking to locate the code which calculates mortgage payments.

The 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, traditional slicing cannot help unless the engineer knows which *variables* are relevant. The engineer may be only partially familiar with the code and, therefore, unable to select a suitable 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 subprogram.

Concept slicing is well suited to this problem. For example, applying the ECS algorithm to the `Calculate:MortgageInterest` concept binding captures the code of interest as a executable sub-program. The `Calculate:MortgageInterest` concept was obtained using the domain model fragment shown in Figure 1 and is highlighted by light shading in Figure 8. Applying the ECS algorithm on `Calculate:MortgageInterest`, identifies the boxed lines in the figure. (Note that the line of code

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

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

The reverse engineer might also analyse the concept binding using Key Statement Analysis. The principal variables of the concept binding are `W-RED-INT-4` and `W-RED-INT`. Using the Bieman-Ott-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:MortgageInterest` concept binding.

Now, suppose instead of applying KSA to the concept binding, the engineer, instead, chooses to apply it to the concept slice. The principal variables of the ECS are `W-RED-INT-4`, `OUT-OUTSTANDING`, `W-RED-INT`, and `APS-RECORD-IN`. For these four variables, the intersection of the corresponding slices is empty, indicating that no statements are *key* in the ECS according to the Bieman-Ott-style algorithm. This information is useful because suggests that more than one concept is implemented within the concept binding. In this case, the ECS happens to contain a large part of the `Read:APSRecord` concept and this should probably be separated out.

At this point, the 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' (a product feature aimed at increasing take up and making the product more attractive by allowing the client to skip a payment for one month, thus 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 binding 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 the previous section, a finer-grained KSA algorithm was also introduced. This approach uses distance 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 binding for mortgage payment. The code segment for the concept binding is cut out and depicted in Figure 7 along with its control-flow graph 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' [40]. What is important is the order they impose 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 'Ball-Eick' style approach.

Finally, suppose that the reverse engineer has extracted several concept bindings (in this example HB-CA reveals 10 concept bindings, but there is insufficient space here to discuss them all in detail). For example, the `Write:APSRecord` concept is shown in the darker shading in the top right-hand column of Figure 8.

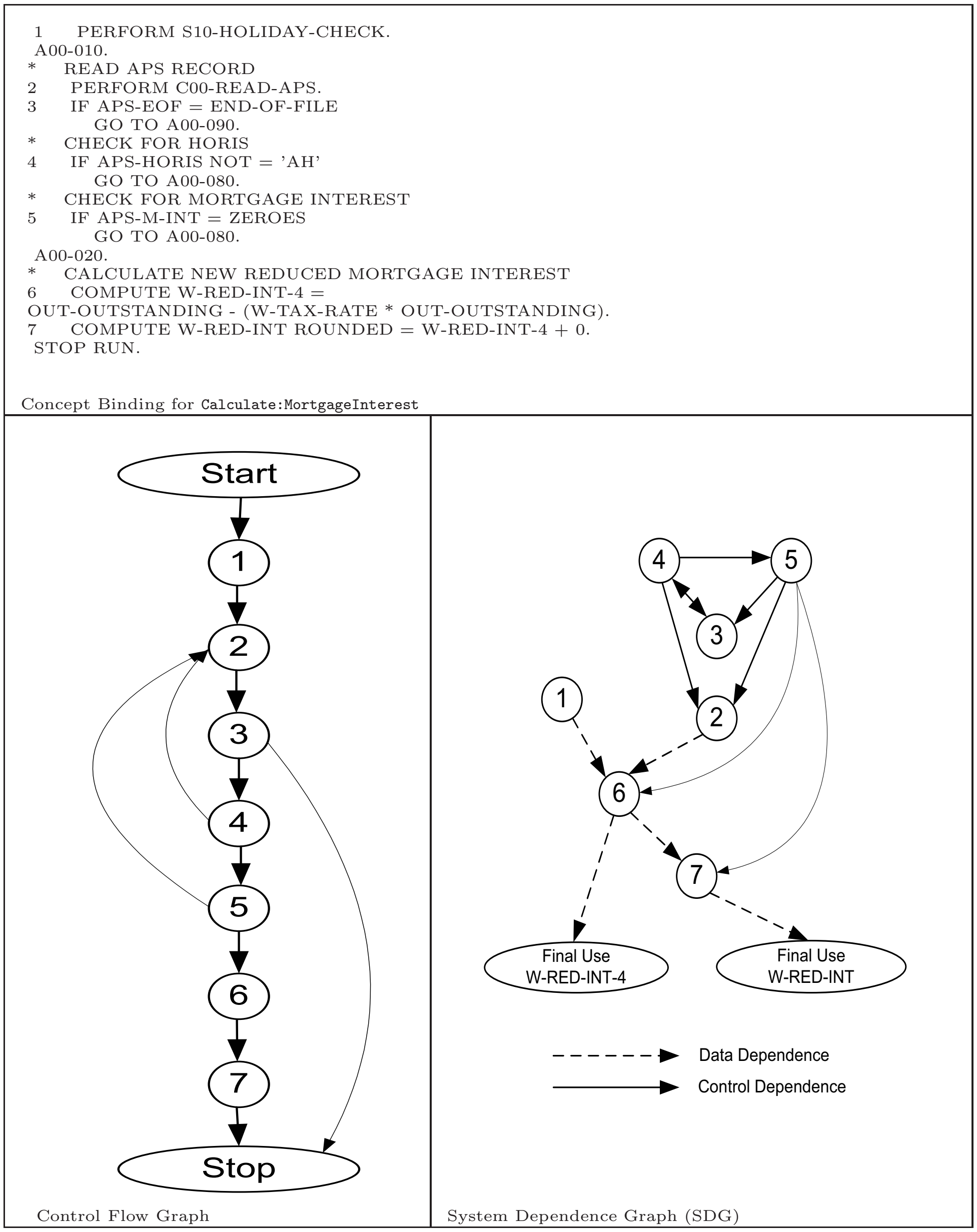
The reverse engineer may be interested in the relationship between this concept binding and the `Calculate:MortgageInterest` concept binding. Such a relationship is useful in refining the domain model, which contains inter-concept relationships. It also provides a crude form of assessment of the impact 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` and `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:MortgageInterest` concept binding to the `Write:APSRecord` concept binding, we face the familiar issue of how to 'count' lines of code [15, 40]. 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:MortgageInterest` concept binding and so the weight of the edge from the `Calculate:MortgageInterest` concept binding to the `Write:APSRecord` concept is $\frac{1}{9}$.

The slice for the principal variables of the `Calculate:MortgageInterest` concept binding consists of the additional boxed lines in Figure 8. We can see that three of these boxed lines are in the `Write:APSRecord` concept binding. However, only one of them is a NCSL, while there are eight NCSLs in total in the `Write:APSRecord` concept binding. This gives the weighting of the relationship from the `Write:APSRecord` concept binding to the `Calculate:MortgageInterest` concept binding as $\frac{1}{8}$.

In summary, this first case study illustrates the way our algorithms are applied using a small COBOL module. It also demonstrates the overall approach and how all four algorithms presented here can be used a part of a concept slicing toolbox. In order to further explore the effects of our approach and to indicate its generality, we now describe a second, much larger, case study using a program from a wholly different domain and written in a different language.



```
 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 Binding for `Calculate:MortgageInterest`

Control Flow Graph

System Dependence Graph (SDG)

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

Case Study 2: UNIX Accounting

Our second case study is a much larger and more complex program written in C. As with the first study, our aim is to demonstrate concept slicing's potential. In this case, we aimed

Key:

Dark Shaded :Write:APSRecord concept
Light Shaded : Calculate:MortgageInterest concept
Boxed : Extra code in ECS for Calculate:MortgageInterest

```
 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.
     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.
```

Figure 8. `PROCEDURE DIVISION` of Cobol Mortgage Payment Calculation Program (continued overleaf)

```
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.
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.
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.
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.
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.
```

to undertake as much of the process automatically as possible. The study uses a version of HB-CA modified to work with C. All slices were constructed using CodeSurfer [21].

Version 6.3.2, of `acct`, a set of programs designed to provide login and process accounting support to GNU/Linux [16] is studied. It consists of 9536 lines of code (as counted by the utility *wc*, including comments and whitespace. Executing `make` on the `acct` suite produces seven executable programs: `ac, last, lastcomm, dump-utmp, dump-acct, accton, sa`.

To extend the generic C concept library for use with *acct*, the source files from the system were examined to identify possible concepts. One file from the system (`ac.c`) was analysed in more depth for likely useful concepts and their evidence. These were added to the library. The application of WeSCA to a new language and domain for which we have minimal domain knowledge meant that an iterative process was necessary to generate suitable, clear bindings for the case study. A number of analyses were thus undertaken across the whole system and the balance of evidence in the library was 'tuned' to ensure that confusion in the concept binder was minimized.

In this case study, consider a software engineer faced with the problem of extracting `acct`'s "user information management processing" for use in another system. In our library, we have created an object concept `UserInfo` with expected evidence such as 'usr', 'account', and 'log' in the comment and identifier classes. The concept has 6 indicators in total. This object is composed with various actions including `Process` and `Write`.

We applied WeSCA to each of the application-related source files in the system. This resulted in 45 concept bindings over 16 analysed files with 4 analysed files having no binding at all (the potential reasons for no binding being made are varied and discussed in more detail in Gold's work [17, 19]). We will assume that all the concept bindings are sufficiently accurate for the engineer to make use of them. The bindings cover 29% of the code (in terms of lines of code assigned) and form a good starting point for our analysis. At face value they look like a good method to focus an engineer's attention upon, since the tool has substantially reduced the amount of code that may be of interest.

The concepts of most interest in this scenario are `Process:UserInfo` and `Write:UserInfo`. Both of these appear to be performing some processing related to the user information needed for use in the new system. These concept binding can be extended using ECS and FCS. First, as `Process:UserInfo` occurs only once in the whole system, this makes it a good starting point. To compute the slices, the appropriate executable from the *acct* suite was built in Codesurfer and slices constructed.

The `Process:UserInfo` concept is bound to the fragment of code from `common.c` shown in Figure 9.

Ideally, forming an executable concept slice on this concept binding would return all the code necessary for processing user information in the system. Using the four lines shown in Figure 9 as the criterion, the backward slice of `sa` includes approximately 0.7% or 33 lines of the source. The code comes from the files `common.c` and `sa.c`. The FCS for this concept binding includes 12 lines of code.

Inspection of the code that makes up these slices shows that two `cpp` macros are key players. In a search for "lower hanging fruit", the engineer might tack a second concept slice based on the `Write:UserInfo` concept. This concept occurs three times in the source code, twice in

```
void
print_acct_file_locations (void)
{
    printf ("The system's default process accounting files are:\n\n");
    printf (" raw process accounting data: %s\n", ACCT_FILE_LOC);
    printf (" summary by command name: %s\n", SAVACCT_FILE_LOC);
}
```

Figure 9. Source code bound for the `Process:UserInfo` concept in `common.c` from the `acct` system [16]

`sa.c`, and once in `dump-acct.c`. Since `sa.c` and `dump-acct.c` occur in two separate builds, our analysis must encompass both.

We begin with the two instances of `Write:UserInfo` found in the file `sa.c`. The code segments nominated by WeSCA for the concepts are shown in Figures 10 and 11. Forming an ECS from the criterion in Figure 10 returns a significant proportion of the code (approximately 24% or 1192 lines). The code returned is spread over all of the application-specific source files. Computing an FCS with this criterion returns approximately 8% (418 lines) of the system over 4 application-specific source files.

The code returned by these operations is substantially concerned with manipulating files and entries within them. It is interesting to note that it includes a large amount of code from `hashtab.c`. Manual analysis indicates that this file is concerned with managing hash tables. WeSCA extracted no concepts from this file so the addition of slicing has proved useful in identifying this aspect of the computation. The engineer now has the code (which for the ECS is executable) for manipulating files of user account information and managing the entries in an internal hashtable data structure.

At this stage we have analysed only one of the `Write:UserInfo` concept bindings in `sa.c`. Computing the ECS from the other instance in `sa.c` (shown in Figure 11) returns the slightly more code (approximately 27% or 1321 lines). The FCS is slightly smaller than the previous example. It includes 7% or 328 lines of code.

The concept slice from the final occurrence of the concept in `dump-acct` returns a slice of 18% of the programs (341 lines). In this case, the FCS is larger than the ECS (which was not the case above) at 29% (or 534 lines). In both cases, the slice extends across all the application-specific source files in the build. The results of the analysis undertaken in this and the previous section are summarised in Table II.

We next consider the intersection of the ECS Results. Since both of the concept bindings from `sa.c` have the same tag and appear to be concerned with similar aspects of the program, it is interesting to determine the extent to which they retrieve the same code from the rest of the system when the ECS is computed. Taking an intersection of the two ECS slices leaves 24% (1178 lines of code). This is very close to the size of either original ECS so it is clear that there is a great deal of common code between the two (as one would hope for two similarly-tagged concept bindings). To further extend this analysis, it is interesting to intersect this result with the slice taken with respect to the `Process:UserInfo` concept. The result is very

```
void
write_savacct_file (char *filename)
{
  struct hashtab_order ho;
  struct hashtab_elem *he;
  FILE *fp;
  char *s;

  /* Write to a temporary file in case we get interrupted. */

  s = (char *) alloca (sizeof (char) * (strlen (filename) + 2));
  sprintf (s, "%s ", filename);

  if ((fp = file_open (s, 1)) == NULL)
    exit (1);

  /* Write each record. */

  for (he = hashtab_first (command_table, &ho);
       he != NULL;
       he = hashtab_next (&ho))
  {
    struct command_key *ck = hashtab_get_key (he);
    struct command_data *cd = hashtab_get_value (he);
    struct savacct sa;

    memcpy (&(sa.c), ck, sizeof (sa.c));
    memcpy (&(sa.s), &(cd->s), sizeof (sa.s));

    if (fwrite (&sa, sizeof (sa), 1, fp) == 0)
    {
      printf ("%s (write_savacct_file): probs writing to file ‘%s’\n",
              program_name, s);
      exit (1);
    }
  }
}
```

Figure 10. Source code bound for the `Write:UserInfo` concept (1st instance) in `sa.c` from the `acct` system [16]

small (0.31% or 16 lines). These lines are mostly cases from the main routine of the program to do with handling files and printing information; effectively a bare outline of the processing

```
void
write_usracct_file (char *filename)
{
  struct hashtab_order ho;
  struct hashtab_elem *he;
  char *s;
  FILE *fp;

  /* Write to a temporary file in case we get interrupted. */

  s = (char *) alloca (sizeof (char) * (strlen (filename) + 2));
  sprintf (s, "%s ", filename);

  if ((fp = file_open (s, 1)) == NULL)
    exit (1);

  /* Write each record. */

  for (he = hashtab_first (user_table, &ho);
       he != NULL;
       he = hashtab_next (&ho))
  {
    char *name = hashtab_get_key (he);
    struct user_data *ud = hashtab_get_value (he);
    struct usracct ua;

    strncpy (ua.name, name, NAME_LEN);
    memcpy (&(ua.s), &(ud->s), sizeof (ua.s));

    if (fwrite (&ua, sizeof (ua), 1, fp) == 0)
    {
      printf ("%s (write_usracct_file): probs writing to file ‘%s’\n",
              program_name, s);
      exit (1);
    }
  }
}
```

Figure 11. Source code bound for the `Write:UserInfo` concept (2nd instance) in `sa.c` from the `acct` system [16]

related to that discovered from the `Write:UserInfo` criteria. Finally, we take the intersection of all three slices formed from the `Write:UserInfo` concept bindings. When this is computed,

Table II. Summary of results for concept slices generated in the second case study

File	Concept	Proportion of System		Line Count Equivalent	
		ECS	FCS	ECS	FCS
`common.c`	`Process:UserInfo`	0.7%	0.2%	33	12
`sa.c`	`Write:UserInfo` (1st instance)	24%	8%	1192	418
`sa.c`	`Write:UserInfo` (2nd instance)	27%	7%	1321	328
`dump-acct.c`	`Write:UserInfo`	18%	29%	341	534

it returns a small set of code; approximately 3% (or 165 lines) of the `sa` build, concerned with manipulating files.

It is possible to refining the above analysis using KSA one of the concept bindings. Using the concept binding shown in Figure 10 as an example, we first identify the principal variables. In this case, the principal variables are `program_name` and `s` because they are parameters to the output statements of the concept binding. Using KSA$_B O$, the engineer would find that almost every statement in the concept contributes to the computation of both principal variables. The key statement analysis results from Codesurfer are shown in Figure 12. It may be desirable to compute KSA$_B E$ in order to further refine the analysis. Unfortunately, this is not possible in our current implementation.

Finally, we consider concept dependency and overlap. We have employed the basic ECS, FCS, and KSA algorithms to determine the source code that should fulfill much of the requirements for extracting user information processing from `acct`. If further analysis were desirable, we could investigate the extent to which the slices already computed overlap with concepts bound elsewhere in the source files.

In a sense, this is a crude form of CDA using un-normalised contribution metrics. Concepts that are overlapped by slices and that seem relevant to the line of investigation may be used as new concept criteria for further analysis. To illustrate this approach, consider the ECS formed from the `Process:UserInfo` concept in `common.c` from program `sa`. We have condensed the results from the various source files impacted by the slice into Table III. Line counts are based on one C statement per line where possible. Preprocessor directives are not counted if they control the inclusion of partial C statements but we do count those controlling the inclusion of whole statements. The results should therefore be considered somewhat approximate. Of these concepts, the only one that appears to be directly relevant to our needs (*i.e.*, the extraction of user information processing) is `Parse:acctfile` and its overlap is not strong by comparison to the more generic `Sort` concept. A second example may be found using the ECS formed from the criterion shown in Figure 10. The overlap is shown in Table IV.

In this example, the `Parse:acctfile` concept occurs again so this would be an appropriate candidate for beginning further analysis with a new ECS at the various points it occurs in the build. As we have already demonstrated the process of computing an ECS, we shall not pursue this further. The most interesting result of this analysis is the amount of slice code that does not occur in any concept binding at all (*e.g.*, 111 lines of this category are from the file

```
void
write_savacct_file (char *filename)
{
  struct hashtab_order ho;
  struct hashtab_elem *he;
  FILE *fp;
  char *s;

  /* Write to a temporary file in case we get interrupted. */

  s = (char *) alloca (sizeof (char) * (strlen (filename) + 2));
  sprintf (s, "%s ", filename);

  if ((fp = file_open (s, 1)) == NULL)
    exit (1);

  /* Write each record. */

  for (he = hashtab_first (command_table, &ho);
       he != NULL;
       he = hashtab_next (&ho))
  {
    struct command_key *ck = hashtab_get_key (he);
    struct command_data *cd = hashtab_get_value (he);
    struct savacct sa;

    memcpy (&(sa.c), ck, sizeof (sa.c));
    memcpy (&(sa.s), &(cd->s), sizeof (sa.s));

    if (fwrite (&sa, sizeof (sa), 1, fp) == 0)
    {
      printf ("%s (write_savacct_file): probs writing to file '%s'\n",
              program_name, s);
      exit (1);
    }
  }
}
```

Figure 12. Source code bound for the `Write:UserInfo` concept (1st instance) in `sa.c` from the `acct` system [16] with results of Key Statement Analysis underlined

Table III. Extent of overlap between slices and concepts for `Process:UserInfo` concept in `common.c`

Concept	Overlapping Lines
`Initialise:Table`	2
`Sort`	35
`Parse:acctfile`	5
No Overlapping Concept	1

Table IV. Extent of overlap between slices and concepts for `Write:UserInfo` concept shown in Figure 10

Concept	Overlapping Lines
`Open:File`	6
`Read:File`	40
`Initialise`	1
`Initialise:Table`	6
`Sort`	29
`Parse:acctfile`	30
`Update:Time`	22
No Overlapping Concept	219

`hashtab.c`). The overlap analysis allows us to identify the extent to which a particular file is related to the concept being analysed but for which there may be no concepts in our domain model. It is also interesting to note that this `Write:UserInfo` concept slice does not overlap with the other occurrence of this concept in the `sa.c` file. This would suggest that the two concepts represent the 'end-points' of their respective computations and one is not subsidiary to the other.

The final part of this case study computes a CDG. Figure 13 shows a part of the CDG based on the ECS formed from the principal variables of the criterion in Figure 10 (`Write:UserInfo`) under the `sa` build). Concepts that do not contribute to the principal variables are not included in the graph. In this case, because CDA operates on concept bindings (rather than simply concept tags), we have explicitly separated each instance of a concept rather than aggregating the metric for each tag. We count lines in the same manner as for the overlap analysis above. The numbers themselves are not as critical as the principles they show.

In summary, these two case studies illustrate the application of our framework to two programs written in different languages. The COBOL case study illustrates the approach on a small program where the operation of the algorithms can be clearly explained in detail. The C case study explores the applicability of the algorithms to a larger program.

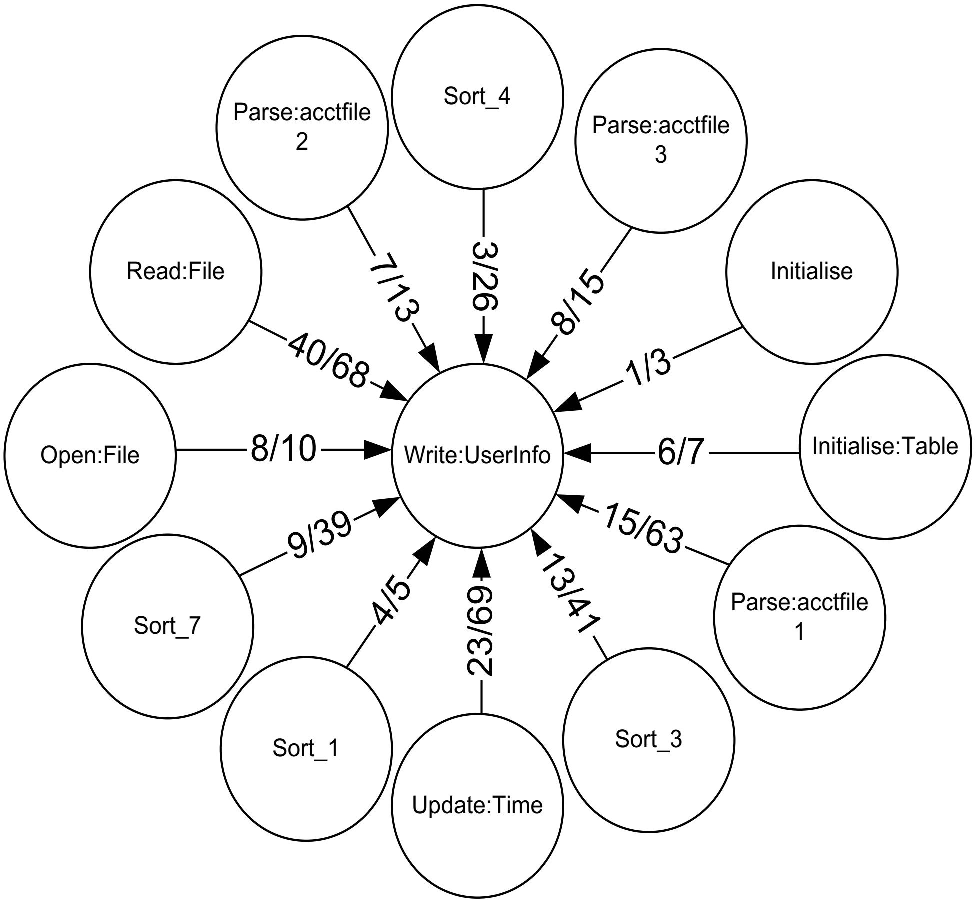


Figure 13. Concept Dependency Graph for the `Write:UserInfo` concept shown in Figure 10

Observations and Lessons Learned

Concept Assignment Performance

HB-CA was slightly more accurate in the COBOL case study than in the C case study but this is to be expected as it originally targeted the COBOL language. A number of unexpected issues were discovered in applying HB-CA to C. HB-CA should be expected to operate successfully on any language similar to COBOL (*i.e.*, imperative languages) [17]. It was not expected that block-structuring in languages like C would pose any particular problem. This proved to be true at a purely syntactic level; C was not substantially harder to perform hypothesis generation on than COBOL. Some differences arose from the way in which the language is used in terms of both the construction of applications (multiple source files) and the use of variables (short identifiers) and comments. These observations lead us to believe that HB-CA's performance could be further improved if the algorithms took more account of language specific features.

Concepts as Slicing Criteria

The concepts we found proved to be useful starting points for analysis. It is interesting to note that, in most of the cases described herein, the backward ECS was much larger than the FCS constructed from the same criterion. This would indicate that the concept assigner was finding high-level concepts towards the end of the computations to which they refer. If we combine this information with observations of concept and slice overlap, it may be possible to form a loose hierarchy of concepts that would be valuable in refining the domain model through improved specialization and composition relationships.

Concept dependency analysis also has the potential to help in this respect by providing the opportunity to cluster concepts based on the weighted graph. This may allow the model to be refined by pruning the list of concepts down to those found regularly in a particular system under analysis. The disadvantage of a pruned library is that HB-CA cannot identify concepts that it has not been told about. Therefore, it is important to gain information about parts of the system to which concepts were not assigned. This issue was demonstrated clearly by the inclusion of the code in hashtab.c by the slicer in our second case study.

Utility of Algorithms

Of the algorithms presented, ECS appears to be the most useful. In most cases, it returns the largest amount of code yet does not return more than is typical for slicing generally [8]. This indicates that the criteria being identified by HB-CA are appropriate. The FCS variation is useful for completeness of extraction but based on the results here, is perhaps less useful in isolation. The potential of KSA is shown best in the first case study where the detail can be seen. It is not clear at this stage whether the substantial amount of code returned for KSA in the second case study is typical. In addition, the version KSA$_{BE}$, requires more sophisticated tool support and then future experimentation.

Conclusions

This paper presents a framework for combining concept assignment and program slicing to increase the effectiveness of both techniques. These two combined provide a software engineer greater leverage than either technique could in isolation. Using the HB-CA approach to concept assignment, four techniques for concept slicing have been proposed. Algorithms for each of these are also presented. We have defined algorithms for refining concept slices to provide a greater level of information to a software engineer. Finally, the algorithms have been implemented and then illustrated through two case studies using a COBOL and a C program.

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. Dave Binkley is supported by National Science Foundation grant CCR-0305330. The authors wish to thank GrammaTech Inc. (`http://www.grammatech.com`) for providing CodeSurfer. This paper appeared, in a preliminary form, in the IEEE Working Conference on Reverse Engineering (WCRE) 2002 [23]. The authors also wish to thank Kathy Newstead for help with presentation of the paper. **REFERENCES**

1. Anquetil, N., and Lethbridge, T. Extracting concepts from file names; a new file clustering criterion. In 20th *IEEE International Conference and Software Engineering (ICSE 1998)* (Kyoto, Japan, Apr. 1998), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 84–93.
2. Ball, T., and Eick, S. G. Visualizing program slices. In *Proceedings of the Symposium on Visual Languages* (Los Alamitos, CA, USA, Oct. 1994), A. L. Ambler and T. D. Kimura, Eds., IEEE Computer Society Press, pp. 288–295.
3. Bieman, J. M., and Ott, L. M. Measuring functional cohesion. *IEEE Transactions on Software Engineering 20*, 8 (Aug. 1994), 644–657.
4. Biggerstaff, T. J., Mitbander, B., and Webster, D. 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.
5. Biggerstaff, T. J., Mitbander, B., and Webster, D. Program understanding and the concept assignment problem. *Communications of the ACM 37*, 5 (May 1994), 72–82.
6. Binkley, D. W. Precise executable interprocedural slices. *ACM Letters on Programming Languages and Systems 3*, 1-4 (1993), 31–45.
7. Binkley, D. W., and Gallagher, K. B. Program slicing. In *Advances in Computing, Volume 43*, M. Zelkowitz, Ed. Academic Press, 1996, pp. 1–50.
8. Binkley, D. W., and Harman, M. A large-scale empirical study of forward and backward static slice size and context sensitivity. In *IEEE International Conference on Software Maintenance (ICSM 2003)* (Amsterdam, Netherlands, Sept. 2003), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 44–53.
9. Binkley, D. W., and Harman, M. A survey of empirical results on program slicing. *Advances in Computers* (2004). To appear.
10. Canfora, G., Cimitile, A., and De Lucia, A. Conditioned program slicing. In *Information and Software Technology Special Issue on Program Slicing*, M. Harman and K. Gallagher, Eds., vol. 40. Elsevier Science B. V., 1998, pp. 595–607.
11. Cimitile, A., Fasolino, A. R., and Marascea, P. Reuse reengineering and validation via concept assignment. In *Proceedings of the International Conference on Software Maintenance 1993* (Sept. 1993), IEEE Computer Society Press, pp. 216–225.
12. Danicic, S., Fox, C., Harman, M., and Hierons, R. M. The ConSIT conditioned slicing system. *Software Practice and Experience* (2004). Accepted for publication.
13. De Lucia, A. Program slicing: Methods and applications. In 1st *IEEE International Workshop on Source Code Analysis and Manipulation* (Florence, Italy, 2001), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 142–149.
14. Devanbu, P., Brachman, R. J., Selfridge, P. G., and Ballard, B. W. LaSSIE: A knowledge-based software information system. *Communications of the ACM 34*, 5 (May 1991), 35–49.
15. Fenton, N. E. *Software Metrics: A Rigorous Approach*. Chapman and Hall, 1990.
16. GNU. http://www.gnu.org/directory/acct.html, 2003.
17. Gold, N. E. *Hypothesis-Based Concept Assignment to Support Software Maintenance*. PhD Thesis, Department of Computer Science, University of Durham, 2000.
18. Gold, N. E. Hypothesis-based concept assignment to support software maintenance. In *IEEE International Conference on Software Maintenance (ICSM'01)* (Florence, Italy, Nov. 2001), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 545–548.

19. Gold, N. E., and Bennett, K. H. Hypothesis-based concept assignment in software maintenance. *IEE Proceedings - Software*. Submitted.
20. Gold, N. E., and Bennett, K. H. A flexible method for segmentation in concept assignment. In 9th *IEEE International Workshop on Program Comprehension (IWPC'01)* (Toronto, Canada, May 2001), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 135–144.
21. Grammatech Inc. The codesurfer slicing system, 2002.
22. Harman, M., Binkley, D. W., and Danicic, S. Amorphous program slicing. *Journal of Systems and Software 68*, 1 (Oct. 2003), 45–64.
23. Harman, M., Gold, N., Hierons, R. M., and Binkley, D. W. Code extraction algorithms which unify slicing and concept assignment. In *IEEE Working Conference on Reverse Engineering (WCRE 2002)* (Richmond, Virginia, USA, Oct. 2002), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 11 – 21.
24. Harman, M., and Hierons, R. M. An overview of program slicing. *Software Focus 2*, 3 (2001), 85–92.
25. Harman, M., Hu, L., Munro, M., Zhang, X., Binkley, D. W., Danicic, S., Daoudi, M., and Ouarbya, L. Syntax-directed amorphous slicing. *Journal of Automated Software Engineering 11* (Jan. 2004), 27–61.
26. Harman, M., Okunlawon, M., Sivagurunathan, B., and Danicic, S. Slice-based measurement of coupling. In 19th *ICSE, Workshop on Process Modelling and Empirical Studies of Software Evolution* (Boston, Massachusetts, USA, May 1997), R. Harrison, Ed.
27. Horwitz, S., Reps, T., and Binkley, D. W. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems 12*, 1 (1990), 26–61.
28. Karakostas, V. Intelligent search and acquisition of business knowledge from programs. *Journal of Software Maintenance: Research and Practice 4* (1992), 1–17.
29. Kohonen, T. *Self-Organizing Maps*, 2nd ed. Springer Series in Information Sciences. Springer-Verlag, Berlin Heidelberg, 1997.
30. Kohonen, T., Hynninen, J., Kangas, J., and Laaksonen, J. Som_pak: The self-organizing map program package. Technical report, Helsinki University of Technology, Laboratory of Computer and Information Science, 1996.
31. Korel, B., and Laski, J. Dynamic program slicing. *Information Processing Letters 29*, 3 (Oct. 1988), 155–163.
32. Longworth, H. D., Ott, L. M., and Smith, M. R. The relationship between program complexity and slice complexity during debugging tasks. In *Proceedings of the Computer Software and Applications Conference (COMPSAC'86)* (1986), pp. 383–389.
33. Maletic, J. I., and Marcus, A. Supporting program comprehension using semantic and structural information. In 23rd *International Confernece on Software Engineering (ICSE 2001)* (Toronto, Canada, May 2001), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 103–112.
34. Merlo, E., McAdam, I., and Mori, R. D. Feed-forward and recurrent neural networks for source code informal information analysis. *Journal of Software Maintenance and Evolution 15* (2003), 205–244.
35. Ott, L. M. Using slice profiles and metrics during software maintenance. In *Proceedings of the* 10th *Annual Software Reliability Symposium* (1992), pp. 16–23.
36. Ott, L. M., and Bieman, J. M. Program slices as an abstraction for cohesion measurement. In *Information and Software Technology Special Issue on Program Slicing*, M. Harman and K. Gallagher, Eds., vol. 40. Elsevier, 1998, pp. 681–699.
37. Ott, L. M., and Thuss, J. J. The relationship between slices and module cohesion. In *Proceedings of the* 11th *ACM conference on Software Engineering* (May 1989), pp. 198–204.
38. Ott, L. M., and Thuss, J. J. Slice based metrics for estimating cohesion. In *Proceedings of the IEEE-CS International Metrics Symposium* (Baltimore, Maryland, USA, May 1993), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 71–81.
39. Rich, C., and Waters, R. C. *The Programmer's Apprentice*. ACM Press (Frontier Series), 1990.
40. Shepperd, M. J. *Foundations of software measurement*. Prentice Hall, 1995.
41. Tip, F. A survey of program slicing techniques. *Journal of Programming Languages 3*, 3 (Sept. 1995), 121–189.
42. Weiser, M. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, MI, 1979.
43. Weiser, M. Program slicing. *IEEE Transactions on Software Engineering 10*, 4 (1984), 352–357.
44. Wills, L. M. *Automated Program Recognition by Graph Parsing*. PhD Thesis, AI Lab, Massachusetts Institute of Technology, 1992.