An Empirical Study of Executable Concept Slice Size

David Binkley
Loyola College
Baltimore MD
21210-2699, USA.

Nicolas Gold, **Mark Harman**, **Zheng Li** and **Kiarash Mahdavi**
King's College London
Department of Computer Science
Strand, London WC2R 2LS, England, United Kingdom.

Abstract

An Executable Concept Slice *extracts from a program an executable subprogram that captures the semantics of a specified high-level concept from the program. Executable concept slicing combines the executability of program slicing, with the expressive domain level criteria of concept assignment. This paper presents results from an investigation of executable concept slice size to assess the effectiveness of executable concept slicing. The results show that the coherence of concept-based slicing criteria allows them to produce smaller executable concept slices than arbitrary criteria, providing evidence for the applicability of Executable Concept Slicing.*

Keywords: Program Slicing, Concept Assignment

1 Introduction

The simplification offered by program slicing has led to its application in a number of reverse engineering tasks, for example, debugging [1, 22, 27, 34], restructuring [12], reuse [4, 13, 26] and program comprehension [14]. One drawback shared by many slice based techniques is the low level at which a slice's starting point (called a slicing criterion) is specified. Essentially, an engineer has to have significant internal knowledge about the code (in particular the program's variables and the locations of their use) before slicing can be effectively employed. For a programmer familiar with a code base, this may not be a problem, but for others it is. Programmers new to the code would benefit from a "higher level" way to specify slicing criterion. Thus, to further broaden the applicability of slicing, one goal of recent work has been to raise the abstraction of level of slice criterion [20].

An *Executable Concept Slice* (ECS) unifies two existing code extraction techniques [15]: concept assignment and program slicing. The result extracts an executable subprogram, based on a high-level conceptually-oriented criterion.

For example, consider an accounting program whose transaction processing code is faulty. The high-level criterion of interest (*transaction processing*) is apparent. But where in the code one might slice to extract the transaction processing code requires in-depth source code inspection. Concept assignment [15] (described below) maps a high-level criterion such as "transaction processing" to a contiguous sequence of source statements. An executable concept slice, for domain level criterion, C, is constructed by forming the distributed union of the slices upon each of the statements identified by the concept assignment binding for C. The union of such slices is an executable program; thus, the resulting union is an executable program that contains the statements required to support the computation identified by concept assignment. This combination of concept assignment and program slicing is referred to as *Executable Concept Slicing* [20].

The applicability of any code extraction technique for reverse engineering crucially depends upon the size of the subprogram extracted. From this point of view, one might be forgiven for supposing that Executable Concept Slicing is simply impractical, because the unioning of several slices will tend to extract large amounts of code; perhaps the entire program in many cases. As this paper demonstrates, this is not the case. The paper presents evidence that the coherence of the slicing criteria identified by concept assignment lead to a set of slices with a large intersection; there is no size explosion when unions of slices for concept-based criteria are constructed.

The three primary contributions of the paper are the empirical evidence provided that

1. Using concept assignment to choose conceptually related slicing criteria produces smaller slices than other similarly-sized criteria.
2. For random collections of slicing criterion there is a positive correlation between the size of the criterion and the size of the resulting slice. In contrast, for conceptually related criterion no such relation exists.

Thus, a concept of n statements does not necessarily lead to a slice that is n times larger than the slice taken with respect to any one of the statements.

3. Large dependence clusters [7] impact the size of slices constructed from all types of slice criteria.

The remainder of the paper is organized as follows. The next section provides background information on program slicing, concept assignment, and their combination in ECS. Section 3 introduces the empirical study undertaken. Section 4 presents and discusses the result of the study. The final sections discuss related work and the conclusions drawn.

2 Background

2.1 Program Slicing

A static backward program slice extracts from a program an executable subprogram composed of the statements relevant to the computation of particular variable v at statement s [32, 33]. Running the program and its slice, the same sequence of values for v are computed at statement s.

The pair $< s, v >$ is referred to as a *slicing criterion*. The criterion can easily be extended to include set of statements and a set of variables at each statements. The resulting slice is the union of the slices taken with respect to the criterion consisting of a single statement and a single variable.

This paper is concerned with static backward slices [32, 33]. For example, Figure 1 shows a simple program that computes the sum and product of the first n integers and the slice highlighted in bold taken with respect to the final value of the the variable `sum`.

Many other forms of slicing have been proposed. These include dynamic slicing [24], amorphous slicing [18, 19] and conditioned slicing [10]. In the remainder of this paper only backward slices are considered [21].

The executable static slice used herein can be computed from a program's *System Dependence Graph* (SDG) as the solution to a graph reachability problem [21, 8]. In doing so, the slicing criterion is simplified to an SDG vertex. The variables from the slicing criterion are assumed to be those referenced by the SDG vertex.

2.2 Concept Assignment

Executable concept slices extend program slicing by replacing the low-level criterion used in slicing with a higher-level criterion. The technique used to identify concepts (*i.e.*, map high-level criterion to low-level criterion) acts as a parameter of the executable concept slicing algorithm. This section describes the use of *Hypothesis-Based Concept-Assignment* to identify the mapping [15]. Alternatives include, for example, a latent semantic indexing based approach similar to the work of Maletic and Marcus [28], and Antoniol et al. [3].

```
sum = 0;
product = 1;
scanf("%d", &n);
for(i=0; i<n; i++)
{
  sum += i;
  product *= i;
}
printf("%d", product);
printf("%d", sum);
```

Figure 1. A example slice taken with respect to the variable sum at the last statement.

Concept assignment aims to allocate specific high-level meaning to specific parts of a program. This technique can be used to identify program fragments associated with a particular concept in a program. For example, the code associated with computing the tax on a sale. The concept assignment problem was defined by Biggerstaff et al. as follows [5]:

> Concept assignment is a process of recognizing concepts within a computer program and building up an "understanding" of the program by relating the recognized concepts to portions of the program, its operational context and to one another.

Hypothesis-Based Concept Assignment (HB-CA) [15] is a plausible-reasoning approach that addresses the first part of Biggerstaff et al.'s concept assignment problem (recognising concepts and relating them to portions of the program).

HB-CA is a three-stage process, *Hypothesis Generation, Segmentation*, and *Concept Binding*. The process requires a knowledge base of concepts to drive the analysis. This is a semantic network that contains two node types: the *concepts* that the programmer may be interested in and the *indicators* that may indicate the presence of a concept in the program (*e.g.*, words that might be found in the source code). The knowledge base is constructed by a software engineer in advance of executing HB-CA (this is not a particularly difficult task since the relationships between concepts and indicators are quite simple). The concept may take the form of an *Action* or *Object* and the indicators can be possible string fragments of identifiers, keywords, or comments.

In the hypothesis-generation stage, the source code is taken as input and scanned for indicators of the various concepts in the knowledge base. A hypothesis to a concept is generated for each matching indicator. All hypotheses are sorted by their indicator position in the source code.

Concept	Source
sum	**sum = 0;**
sum	**for(i=0; i<N; i++**
sum	**sum += Sound[i];**
spl	effective_SPL = 20×log($\sqrt{\text{sum}}$)
average	***average = sum / N***

Figure 2. A simple HB-CA example that includes three concepts related to sound volume. Sound pressure level (SPL) describes the relative intensity of a sound.

In the segmentation stage, the hypothesis list is analysed to group hypotheses into segments, initially using natural segment boundaries such as procedure boundaries. Following this, a *self-organising map* can be used to create non-overlapping partitions of high conceptual focus [15]. The output of this stage is a collection of segments, each containing a number of hypotheses.

In the concept-binding stage, the segments are scored in terms of concept occurrence frequency. Disambiguation rules select a concept where several different hypotheses occur equally frequently. The result of this stage is a series of concept bindings labeled with the winning concepts accordingly.

Figure 2 provides a simple HB-CA example in which the domain model includes indicators for three concepts: sum indicates the computation of the sum of a collection of sound frequencies, spl, pressure, level indicate the computation of the effective sound pressure level of the sound, and average indicates the computation of the average volume of the sound's frequencies. The first and third statement include an indicator for the *sum* concept. The penultimate statement includes an indicator for the *spl* concept, and the final statement includes an indicator for the *average* concept and the sum concept. In this highly-simplified example, Lines 1-3 are assigned to *sum* concept, Line 4 to the *spl* concept and Line 5 to the *average*. Note that generally HB-CA assigns to regions rather than single lines and the results of HB-CA are not guaranteed to be executable.

2.3 Executable Concept Slicing

An ECS is formed using slicing to extend the results of concept assignment by slicing with respect to the components associated with a given concept. This computation exploits the advantages of both techniques, while overcoming their individual weaknesses. Executable concept slicing takes advantage of the executability of programs produced by slicing and the high-level extraction criterion from concept assignment. This contrasts with the unexecutable concept binding produced by concept assignment alone and the low-level criterion required for program slicing.

Empirical results based on the study of one COBOL module from a large financial organization and one open-source C program have indicated that combining concept assignment and program slicing produces better results than either is capable of individually [16].

An algorithm for computing all the executable concept slices is presented in Figure 3. For each concept c_i, it computes the slice taken with respect to the set of SDG vertices that represent statements from c. Slicing on these vertices produces the ECS.

function ECS (Program P, DomainModel D)
returns: set of Program

let $\{c_1, \ldots, c_n\} = HB\text{-}CA\text{-}Concepts(P, D)$
for *each* $c_i \in \{c_1, \ldots, c_n\}$
  let $ECS_i = Slice(P, SDG\text{-}vertices(c_i))$
endfor
return $\{ECS_1, \ldots, ECS_n\}$

Figure 3. The Executable Concept Slicing Algorithm

Reconsider the example shown in Figure 2. An engineer interested in the average sound volume could apply HB-CA. The result is the last statement of the program (shown in ***bold italics***). Slicing on the SDG vertex (in general vertices) for the statements identified by HB-CA returns the vertices represented by the bold statements Figure 2. These statements make up the ECS for the average concept. They form an executable program that computes average volume.

3 Empirical Study

This section describes the setup of the case study used to empirically investigate concept slices. The study compares the size of slices constructed using four different generators of low-level slicing criteria. Each generator produces a set of SDG vertices. For example, the HB-CA generator returns the vertices representing all the statements identified as belonging to a particular concept. The following three research questions are used to capture the most important aspects of concept slice construction.

1. Does slicing on a set of conceptually related SDG vertices (*i.e.*, those representing a single HB-CA concept) produce a smaller slice than slicing on other similarly sized sets?
2. Is the size of an ECS related to the size of the HB-CA extracted criterion?
3. Do large dependence clusters [7] affect ECS size?

3.1 Criteria Definitions

In order to answer these questions four alternate techniques for selecting similarly sized sets are compared. The first of the four, hereafter referred to as *Type 1*, uses the concept binding produced by a concept assigner (in this case the WeSCA implementation of HB-CA [15]). This approach is expected to generate the most coherent sets of criteria and thus the smallest slices.

The remaining three types generate similar sized, but less coherent sets of criteria. Types 2 and 3 select SDG vertices that represent contiguous program statements starting from a random statement; thus, it is unlikely that these statements correspond to a single concept. A Type 2 criterion includes a prescribed number of vertices. In the experiment, this is always the same number as found in the corresponding Type 1 criterion. This type of criterion reflects most closely the structural form of a Type 1 criterion.

A Type 3 criterion includes a prescribed number of lines of code. In the experiment, this is always the same number of lines as found in the corresponding Type 1 criterion. Thus, this type of criterion reflects the lexical structure of a Type 1 criterion.

Finally, Type 4, the least coherent type, includes a random collection of vertices. In the experiment, the size of this set is the same as that of the corresponding Type 1 criterion.

The four types are formally defined below. In the definitions, lines of code include statements, comments, and blank lines, as these are all relevant to concept assignment.

Type 1: Concept Binding

For program P and a concept C generated by a concept assigner, the Type 1 criterion for C includes the vertices of P's SDG that represent the statements of C.

Type 2: Contiguous, Same Vertex Count, Random Start

For a program P, given natural number n and starting statement s_i, a Type 2 criterion includes the first n vertices of P's SDG that represent the statements $s_i, s_{i+1}, \cdots$.

Type 3: Contiguous, Same LoC Count, Random Start

For a program P, given natural number n and starting statement s_i, a Type 3 criterion includes the vertices that represent the n statements $s_i, s_{i+1}, \cdots s_{i+n-1}$.

Type 4: Non-Contiguous, Same Vertex Count, Random Points

For a program P and a given natural number n, a Type 4 criterion is a collection of n randomly selected vertices from P's SDG.

Figure 4 illustrates the four types of criteria. In the figure a statement is represented by a gray line while an SDG vertex by a gray dot. *Selected* components are shown in darker grey. Line 12 separates two concepts. Thus, assuming the concept from Lines 2-11 is of interest then the 17 vertices that represent these 10 lines would make up the Type 1 criterion. The example Type 2 criterion is generated by selected a random starting location (Line 6 in the example) and including contiguous statements until 17 vertices are included (when multiple vertices represent a statement, the starting and ending positions within the line are used to uniquely order the vertices for selection). The example Type 3 criterion is generated by selecting a random starting location (Line 4 in the example) and including contiguous statements until 10 lines are included. Finally, the example Type 4 criterion includes 17 randomly selected vertices.

The primary comparison made in the study is between Type 1 criterion and Type 4 criterion. In essence the question being asked is does using concept assignment to select a set of slicing criteria differ from just selecting random vertices. If there is no different then an ECS is likely to get too large to be useful in practice. It is anticipated that the average ECS size will be similar to the average backward slice. This should be the case if the statements identified as belonging to a single concept are related in a dependence sense.

Comparison of the resulting slices for Type 1 criterion with those for Type 2 and Type 3 allows the effect of conceptual coherence (as defined by the hypothesis-based concept-assignment algorithm) and spatial coherence (adjacent lines of code) to be better understood. If the spatial aspect of Type 1 criterion is all that effects slice size, then slices on Type 2 and Type 3 criterion will have the same size as corresponding Type 1 criterion. Conversely, if conceptual coherence has value then Type 1 criterion will lead to smaller slices.

This difference will never be large because any consecutive sequence of statements has some conceptual coherence. To better understand the relationship, consider, for example, a program that is partitioned into concept bindings each consisting of exactly 10 statements. A randomly chosen sequence of 10 statements has a 10% probability of being identical to one of the existing concept bindings (if its starting point is the starting point of a concept). The other 90% of the time such a randomly selected sequence of 10 statements will include portions of two concepts. If these two concepts are related (in a dependence sense), then the slices on both will include significant overlap. Therefore, it would be realistic to expect that Type 2 and Type 3 criteria would lead to a size increase compared to Type 1 criterion, but it would be unrealistic to expect that there would be a significant difference.

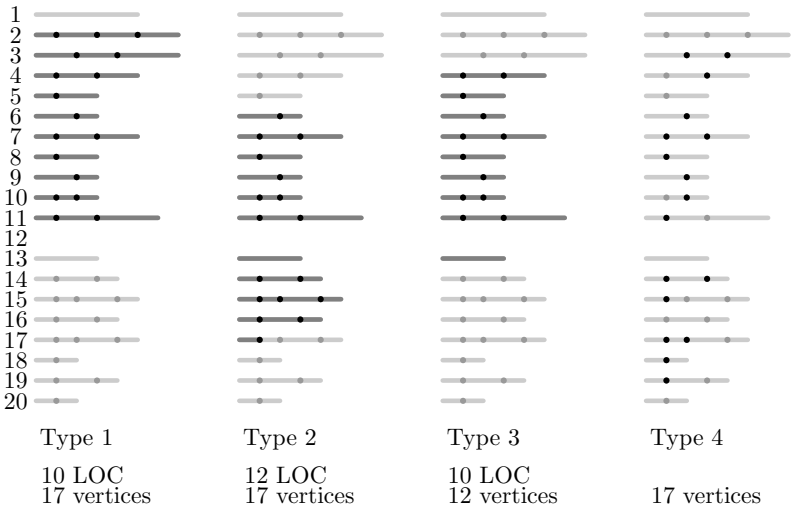

Figure 4. Example of the four types of criteria

Subjects	LoC	Vertices	Number of Concept Bindings	Large Dependence Cluster	Brief Description
acct-6.3.2	3,204	9,775	24	No	Process monitoring utilities
EPWIC-1	7,943	19,545	63	No	Image compressor
space	9,126	20,018	67	No	ESA space program
indent-2.2.6	8,259	30,311	49	Yes	Text Formatter
diffutils-2.8	10,743	33,231	109	Yes	File comparison utilities
findutils-4.2.25	28,887	105,535	128	Yes	File finding utilities
Total	68,162	218,415	440		

Table 1. Experimental Subjects.

3.2 Subject Programs

The six programs used in the study are summarised in Table 1, which lists each program along with some statistics related to the program and its SDG. These include the size of the program in LoC and the size of the resulting SDG in vertices. Both LoC and vertices were counted by CodeSurfer [17]. The table also shows the number of concept bindings that were identified by WeSCA for each subject program. Over all six programs there were a total of 440 concept bindings.

To address the third research question, the programs selected include three subject programs known to be free of large dependence clusters (acct-6.3.2, EPWIC-1, and space) and three subject programs known to contain large dependence clusters (indent-2.2.6, diffutils-2.8, and findutils-4.2.25). Dependence clusters are essentially strongly connected components (SCCs) in the dependence graph. From the standpoint of a dependence analysis, such as slicing, including any part of such an SCC causes the inclusion of the entire SCC.

3.3 Analysis Tools

WeSCA, a tool based on HB-CA and developed by Gold, was used to identify concept bindings in the source programs [16]. WeSCA requires a library of concepts to drive its analysis. A generic C concept library, used in other studies [16], was used as the basis for the analysis. It was extended for each program to reflect the different purposes (and thus concepts) of each program. To extend the generic C concept library, source files from each system were examined to identify possible concepts.

CodeSurfer, Grammatech's deep-structure analysis tool, was used to build the SDGs and compute program slices in the experiments reported herein [17]. The API for CodeSurfer includes a function that identifiers the dependence graph vertices associated with a range of source line numbers. After WeSCA identifies the lines associated with a concept, this API is used to identify the corresponding SDG vertices used as slice starting points. The resulting ECS is the (backward) slice taken with respect to these vertices.

3.4 Measurement

Lines of Code (LoC) is a somewhat crude metric for measuring slice size owing to the lack of a standard definition for LoC. Following Binkley et al., herein size is more precisely measured using SDG vertices [9]. In this experiment, both the size of the low-level slicing criterion and the size of the resulting slices are reported as a percentage of the source-code representing SDG vertices. Excluded from this count are internal vertices that do not directly represent source. The most common source of such vertices is the representation of global variables as additional function parameters.

3.5 Method

The case study was performed by executing WeSCA on the subject programs to generate a set of Type 1 criteria for each program. These were exported (in terms of the start and end line numbers) to data files that were imported into CodeSurfer and mapped into sets of SDG vertices that form the low-level slicing criterion. Finally, slices for each vertex of the set were computed using CodeSurfer. The result of each slicing operation is a set of vertices, and the union of these sets forms the vertices of the ECS.

The size of the Type 1 criterion was used to determine the size of the other three criterion types. For example, consider a Type 1 criterion of 10 LoC that corresponds to 17 SDG vertices. The comparable Type 2 criterion would include 17 SDG vertices taken from a random contiguous sequence statements. Similarly, the comparable Type 3 criterion would include a random contiguous sequence of 10 statements. Finally, the comparable Type 4 criterion would include 17 randomly chosen SDG vertices.

To compare the effect of the different types, for each Type 1 criterion, a corresponding Types 2, 3, and 4 criteria of a matching size to the Type 1 was generated. Since Types 2, 3, and 4 rely on random generation they were repeated 30 times for each Type 1 criterion and the results averaged.

4 Results and Discussion

4.1 Relationship between Criterion Coherence and Slice Size

Figure 5 shows the average slice size for each program and each type. Recall that for Type 2, 3, and 4 criteria the resulting slice size is the average of 30 slices taken from different random starting points. Comparing the slices taken with respect to Type 1 and Type 4 criteria, it is clear from the graph that Type 4 criterion always generates larger slices than the more computationally coherent Type 1 criterion. This is important for two reasons. First, it provides evidence that HB-CA is identifying conceptually related vertices (and thus vertices with similar slices). Second, it provides evidence that slicing from a large set of vertices V

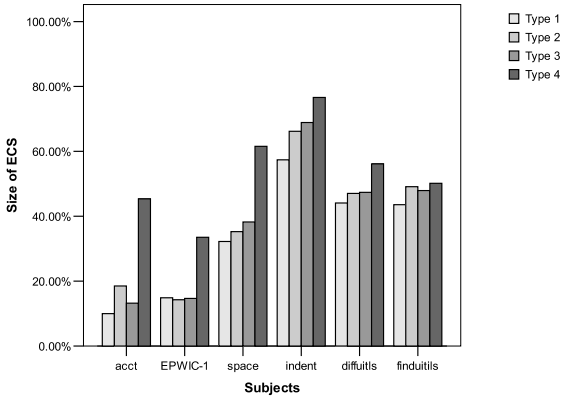

Figure 5. Average slice sizes for each type of criterion.

(*e.g.*, those representing a concept) does not lead to a significant increase in slice size when V has a degree of conceptual coherence. In contrast, as can be seen from the results, slicing on a set of vertices V that are purely random (and therefore have no conceptual coherence) does lead to a significant increase in size.

It is possible that the larger slice size when using Type 4 criterion results from a lack of spatial locality, rather than a lack of conceptual locality. That is, instead of indicating that HB-CA is identifying semantically related program components (and thus their slice is smaller), the results comparing Type 1 and Type 4 might indicate that selecting lexically close program components (spatial locality) produces smaller slices. Type 2 and Type 3 criteria were included in the experiment in order to investigate this possibility. Both Type 2 and Type 3 criteria are built from contiguous program statements and thus should have similar spatial locality with Type 1 criterion.

Using Type 2 or Type 3 criteria provides a very strict test. Although the starting points for both criteria are random, by including contiguous program statements, some contiguous sequence of one or more concepts binding will always be included. Indeed, this is inevitable, since concept assignment partitions the source code into a set of non-overlapping concepts.

Figure 5 provides evidence that Type-1 criterion do lead to smaller size increases than Types 2 and 3. That is, in all but one case, the Type 1 criterion lead to smaller size executable concept slices. Thus, it is conceptual locality and not spatial locality that causes executable concept slices to be smaller those computed using Type 4 criterion.

The one exception is for the program EPWIC where Type 2 criterion produced a smaller average slice size than the average ECS slice. However, the difference is not statistically significant. Finally, it is interesting to note for the programs with large dependence clusters, the average sizes of slices from all criteria show greater similarity.

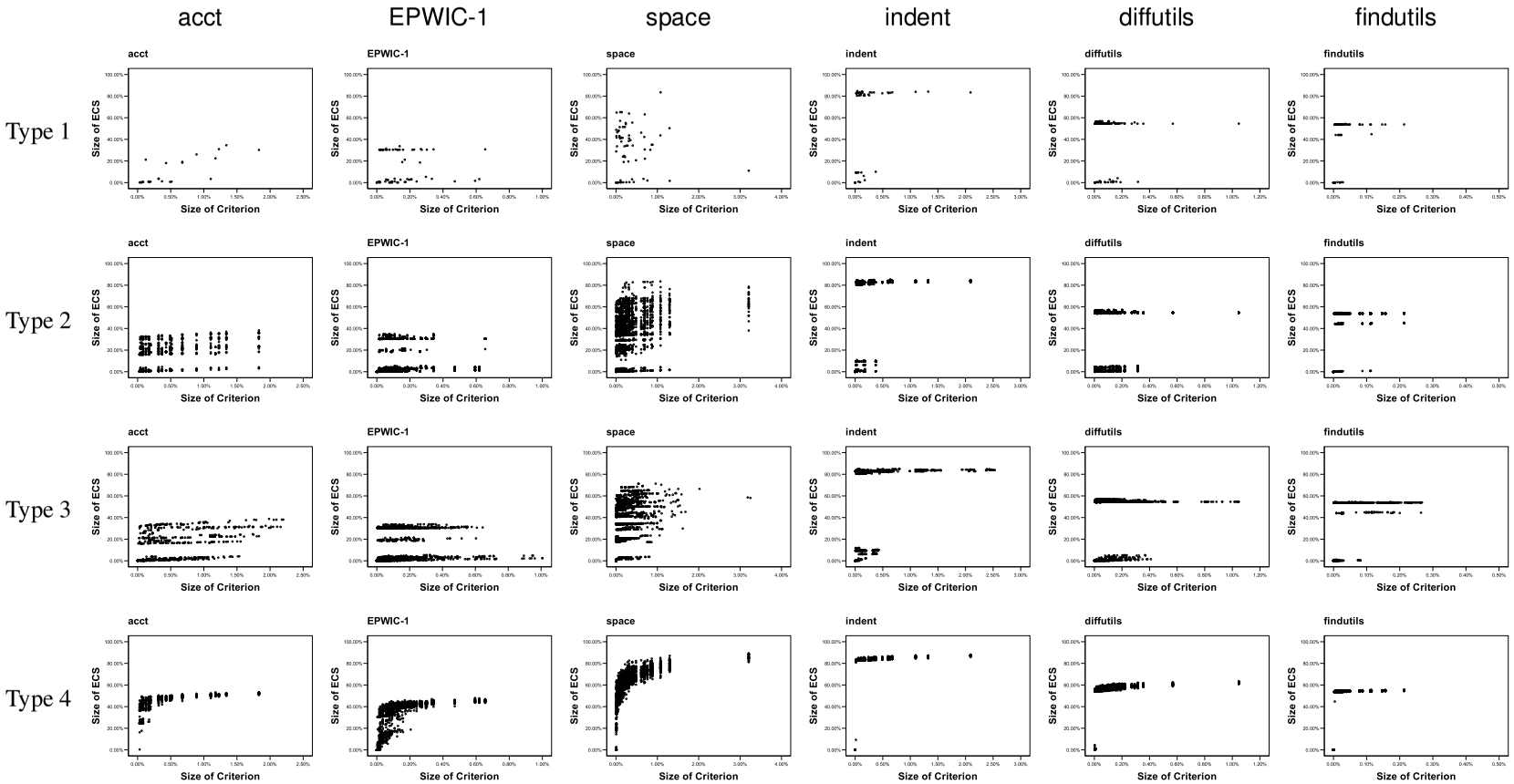

Figure 6. Scatter plots to show the relationship between the size of criterion and the size of slices for all four types of criteria.

4.2 Relationship between Criterion Size and Slice Size

Figure 6 presents scatter plots of the size of the resulting slices for each subject program using all four types of criteria. The columns in the figure represent each program and the rows the criterion types. For each sub-figure, the X axis represents the size of criterion (each program uses a slightly different scale ranging from 0.5% to 4.0% as the largest X value). The Y axis is the corresponding slice size. The detail of each figure is less important than its overall shape. Note that for Types 2, 3, and 4, all slices are shown (*i.e.*, 30 slices corresponding is size to each Type 1 criterion).

Looking first across all four types, there is a clear difference in the pattern for the first three programs and the last three programs. The limited number of bands for the last three programs are caused by the presence of large dependence clusters. To better understand this, consider a program that is one large dependence cluster (one large SCC of dependences). Any slice of such a program will include a component from the cluster and thus will include the entire cluster (entire program). In the more realistic case, the existence of a few significant dependence clusters means that only a limited number of output slice sizes are possible. For example, consider the scatter plot for indent using the Type 4 criterion. Only 121 of the 1470 runs (less the 10%) of the slices managed to miss the one large dependence cluster. (The 121 are similar enough in value to appear as two dots on scatter plot.) As a result the size of most slices is just over 80%. In contrast using Type 1 criterion, over a dozen of the 49 concepts (25%) map to vertices outside this large dependence cluster. As shown in Figure 6 the impact of large dependence clusters is the presence of characteristic horizontal lines in the size graphs indicating. These lines witness the large interdependencies between program components.

Looking at each criterion type independently, the random criterion, Type 4, shows the expected increase in slice size accompanying an increase in criterion size. This effect is masked by large dependence clusters; thus, it is easier to see in the first three programs than in the last three programs. The best example is the space program where the pattern is quite visually apparent. This correlation comes from the probability of a subsequent randomly chosen vertex being in a new area of the program, as vertices are added more of the program is in the slice and thus the probability of including a new area diminishes.

Statistically this kind of non-linear relationship should give rise to a strong Spearman rank correlation, which simply quantifies how frequently an increase in y accompanies an increase in x. The relationship between x and y need not be linear. For acct, EPWIC-1, and space, the data from the Type 4 scatter plots have Spearman R's coefficients of 0.91, 0.86, and 0.87, respectively. All being greater than 0.80 these suggest strong rank correlations.

This non-linearity can be explained if the selection of scattered vertices is seen as a sequential process of adding slices to the union one at a time (and simultaneously increasing the size of the criterion). The first few vertices selected have a much greater chance of drawing in large parts of the program than those selected later (since a large amount of the program has already been drawn in by earlier-selected vertices in the criterion). The last few vertices selected have a much smaller amount of new program left to draw on and thus the size increases as they are added to the criterion is much smaller.

In contrast, with one exception, Type 1 criterion do not exhibit any predictive relationship between criterion size (the number of SDG vertices from which the slice is taken) and slice size. This is visually apparent in the scatter plots, which appear as clouds. The one exception is acct where a linear correlation exists yielding a Pearson's R coefficient of 0.79 (a strong linear relation). The model shows a 19% increase in slice size accompanies a 1% increase in criterion size. This indicates that acct's concepts are essentially nested (from a dependence standpoint) based on size; thus, there is rarely a smaller concept whose vertices depend on the vertices of a larger concept. This relationship holds for the graphed data in which the size of the criterion is less that 2% of the SDG vertices. Clearly this growth is not sustainable for larger criterion sizes.

Owing to the spacial locality of Type 2 and 3 criteria, they include a limited number of concepts (most often 1 or 2). The scatter plots reinforce the results shown in Figure 5. They do show that the averages are not influenced by any outliers and thus are a fair summary of the slices sizes. Finally, comparing the scatter plots for EPWIC-1 using Type 1 and Type 2 criterion the cause of the lower average for Type 2 can be seen. The two plots have similar structure. The key difference in the larger number of small slices for Type 2 (the dark line with y being about zero). For Type 2 criterion there are a slightly larger proportion of criterion that produce these small slices and thus the slightly smaller average.

5 Related Work

Various approaches have been used to undertake concept assignment besides the HB-CA method used here. The DM-TAO system [5, 6] used a rich semantic network as its representation and computation engine to identify source code in comments. IRENE was designed to extract business rules by matching source code information to known rule patterns [23]. More recent work has adopted neural networks [30] and latent semantic indexing [28].

The techniques used in concept assignment are similar to those that apply information processing techniques to software engineering. For example, if one considers the knowl-

edge base to be external program documentation then work on (re)establishing links between the program and the external documentation [3, 28] is similar to the task of identifying which parts of the program represent a particular concept.

Program slicing has been applied in pursuit of executable code extraction (using traditional non-conceptual slicing criteria). Lanubile and Visaggio use slicing to extract reusable components [25]. Ning et al. adopt a two-step process for component extraction using statement identification and bi-directional slicing [31].

Previous work on alternative types of slicing criteria includes that of Canfora et al. [10, 11] on conditioned slicing. Gold et al. [16, 20] described a framework for unifying concept assignment with slicing. Four methods for combining these two analysis techniques were presented as Executable Concept Slicing Forward Concept Slicing, Key Statement Analysis, and Concept Dependence Analysis, respectively. Two cases, a COBOL module based on one from a large financial organization and an open source utility program written in C were used to illustrate the techniques.

Finally, Al-Ekram and Kontogiannis combined concept assignment, formal concept analysis, and program slicing, presenting a program representation formalism-"the Lattice of concept slices" and a program modularization technique that aims to separate statements in a code fragment according to the concept they implement[2].

Though concept assignment derives a high degree of expressibility from the domain level at which its extraction criteria are expressed, it lacks the executability of program slicing. However, slicing, though it produces executable subprograms, can only do this using extremely low level criteria. The present paper presents the first empirical evidence that it is possible to combine the expressive power of these concept assignment techniques with the semantic guarantees that accrue from the executability of program slicing.

6 Conclusions and Future Work

This paper empirically studies executable concept slicing using six subject programs. More than 400 concept bindings were identified and studied. The results from the study provide evidence to support the claim that the statements identified by concept assignment are not arbitrary: they form a coherent set of related parts of the program, because the slices constructed from them have a large degree of overlap. This is an encouraging finding, because it indicates that there is no 'size explosion' when making executable concept slices.

Future work will expand the scope of this study, adopting alternative approaches to generated concept-oriented slice criterion (*e.g.*, Marcus et al.'s LSI-based approaches to concept assignment [29]) and increasing the number of subject programs. The impact of large dependence clusters on this and other work suggests that identifying and handling these artifacts of dependence is an important topic for future work.

Acknowledgements

This research work is supported by EPSRC Grant GR/T22872/01, GR/R71733/01 and by National Science Foundation grant CCR0305330. The authors also wish to thank GrammaTech Inc. (http://www.grammatech.com) for providing CodeSurfer and Greg Rothermel for providing some of the case study programs.

References

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Software Practice and Experience*, 23(6):589–616, June 1993.

[2] R. Al-Ekram and K. Kontogiannis. Source code modularization using lattice of concept slices. In *CSMR*, pages 195–203, 2004.

[3] G. Antoniol, G. Canfora, G. Casazza, and A. DeLucia. Information retrieval models for recovering traceability links between code and documentation. In *Proceedings of 2000 International Conference on Software Maintenance*, San Jose, California, October 2000. IEEE Computer Society Press, Los Alamitos, California, USA.

[4] J. Beck and D. Eichmann. Program and interface slicing for reverse engineering. In *IEEE/ACM* 15^{th} *Conference on Software Engineering (ICSE'93)*, pages 509–518. IEEE Computer Society Press, Los Alamitos, California, USA, 1993.

[5] T. J. Biggerstaff, B. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *15th International Conference on Software Engineering*, pages 482–498, Baltimore, Maryland, United States, May 1993. IEEE Computer Society Press.

[6] T. J. Biggerstaff, B. Mitbander, and D. Webster. Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5):72–82, May 1994.

[7] D. Binkley and M. Harman. Locating dependence clusters and dependence pollution. In 21^{st} *IEEE International Conference on Software Maintenance (ICSM 2005)*, pages 177–186, Budapest, Hungary, 2005. IEEE Computer Society Press, Los Alamitos, California, USA.

[8] D. W. Binkley. Precise executable interprocedural slices. *ACM Letters on Programming Languages and Systems*, 3(1-4):31–45, 1993.

[9] D. W. Binkley and M. Harman. A large-scale empirical study of forward and backward static slice size and context sensitivity. In *IEEE International Conference on Software Maintenance (ICSM 2003)*, pages 44–53, Amsterdam, Netherlands, Sept. 2003. IEEE Computer Society Press, Los Alamitos, California, USA.

[10] G. Canfora, A. Cimitile, and A. De Lucia. Conditioned program slicing. In M. Harman and K. Gallagher, editors, *Information and Software Technology Special Issue on Program*

Slicing, volume 40, pages 595–607. Elsevier Science B. V., 1998.

[11] G. Canfora, A. Cimitile, A. De Lucia, and G. A. D. Lucca. Software salvaging based on conditions. In *International Conference on Software Maintenance (ICSM'96)*, pages 424–433, Victoria, Canada, Sept. 1994. IEEE Computer Society Press, Los Alamitos, California, USA.

[12] 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.

[13] A. Cimitile, A. De Lucia, and M. Munro. A specification driven slicing process for identifying reusable functions. *Software maintenance: Research and Practice*, 8:145–178, 1996.

[14] A. De Lucia, A. R. Fasolino, and M. Munro. Understanding function behaviours through program slicing. In 4^{th} *IEEE Workshop on Program Comprehension*, pages 9–18, Berlin, Germany, Mar. 1996. IEEE Computer Society Press, Los Alamitos, California, USA.

[15] N. E. Gold and K. H. Bennett. Hypothesis-based concept assignment in software maintenance. *IEE Proceedings - Software*, 149(4):103–110, Aug. 2002.

[16] N. E. Gold, M. Harman, D. Binkley, and R. M. Hierons. Unifying program slicing and concept assignment for higher-level executable source code extraction: Research articles. *Softw. Pract. Exper.*, 35(10):977–1006, 2005.

[17] Grammatech Inc. The codesurfer slicing system, 2002.

[18] M. Harman, D. W. Binkley, and S. Danicic. Amorphous program slicing. *Journal of Systems and Software*, 68(1):45–64, Oct. 2003.

[19] 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.

[20] M. Harman, N. Gold, R. M. Hierons, and D. W. Binkley. Code extraction algorithms which unify slicing and concept assignment. In *IEEE Working Conference on Reverse Engineering (WCRE 2002)*, pages 11 – 21, Richmond, Virginia, USA, Oct. 2002. IEEE Computer Society Press, Los Alamitos, California, USA.

[21] 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.

[22] M. Kamkar. *Interprocedural dynamic slicing with applications to debugging and testing*. PhD Thesis, Department of Computer Science and Information Science, Linköping University, Sweden, 1993. Available as Linköping Studies in Science and Technology, Dissertations, Number 297.

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

[24] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, Oct. 1988.

[25] F. Lanubile and G. Visaggio. Extracting reusable functions by flow graph-based program slicing. *IEEE Transactions on Software Engineering*, 23(4):246–259, 1997.

[26] D. Liang and M. J. Harrold. Reuse-driven interprocedural slicing in the presence of pointers and recursion. In *IEEE International Conference of Software Maintenance*, pages 410–430, Oxford, UK, Aug. 1999. IEEE Computer Society Press, Los Alamitos, California, USA.

[27] J. R. Lyle and M. Weiser. Automatic program bug location by program slicing. In 2^{nd} *International Conference on Computers and Applications*, pages 877–882, Peking, 1987. IEEE Computer Society Press, Los Alamitos, California, USA.

[28] J. I. Maletic and A. Marcus. Supporting program comprehension using semantic and structural information. In 23^{rd} *International Confernece on Software Engineering (ICSE 2001)*, pages 103–112, Toronto, Canada, May 2001. IEEE Computer Society Press, Los Alamitos, California, USA.

[29] A. Marcus and J. I. Maletic. Identification of high-level concept clones in source code. In *Proceedings of the 16th International Conference on Automated Software Engineering (ASE 2001)*, pages 107–114, Nov. 2001.

[30] E. Merlo, I. McAdam, and R. D. Mori. Feed-forward and recurrent neural networks for source code informal information analysis. *Journal of Software Maintenance and Evolution*, 15:205–244, 2003.

[31] J. Q. Ning, A. Engberts, and W. Kozaczynski. Recovering reusable components from legacy systems by program segmentation. In *IEEE Working Conference on Reverse Engineering*, pages 64–72, Sorrento, Italy, 1993.

[32] M. Weiser. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, MI, 1979.

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

[34] M. Weiser and J. R. Lyle. *Experiments on slicing–based debugging aids*, chapter 12, pages 187–197. Empirical studies of programmers, Soloway and Iyengar (eds.). Molex, 1985.